

# Погружение в Рефакторинг

❖ Оффлайн Редакция



**REFACTORING**  
· GURU ·

Александр Швец

# Погружение в Рефакторинг

*Оффлайн Редакция (Java)*

v1.0

Книга только для клубного пользования!!!



# Вместо скучного копирайта

Привет! Меня зовут Александр Швец, я автор онлайн курса «Погружение в Рефакторинг» ([www.refactoring.guru/ru/course](http://www.refactoring.guru/ru/course)), частью которого является данная книга.

Эта книга предназначена для вашего личного пользования. Пожалуйста, не передавайте книгу третьим лицам, за исключением членов своей семьи. Если вы хотите поделиться книгой с друзьями или коллегами, то купите и подарите им полную (и легальную) версию курса.

Все деньги, вырученные с продаж курса, идут на развитие Refactoring.Guru. Каждая проданная копия приближает момент выхода нового курса или книги.

© Александр Швец, Refactoring.Guru, 2017

[support@refactoring.guru](mailto:support@refactoring.guru)



# Введение

Я попытался перенести в эту книгу максимум информации, доступной в полном курсе о рефакторинге. По большей части, мне это удалось. Но некоторые вещи, вроде *живых примеров*, попросту невозможно преподнести в рамках статичной электронной книги. Поэтому, воспринимайте эту книгу как вспомогательный материал, а не полную замену курса.

Книга разбита на две большие секции: **Запахи плохого кода и Техники рефакторинга**. В первой секции описано то, как не надо писать код. Во второй – то, как уже написанный плохой код можно улучшить.

Книгу можно читать как последовательно, от края до края, так и в произвольном порядке. Несмотря на то, что все темы тесно переплетены друг с другом, вы сможете с лёгкостью прыгать по связанным темам, используя ссылки, которых в книге имеется в достатке.

Примеры в этом варианте книги приведены на языке **Java**. Остальные варианты можно скачать, зайдя в свой аккаунт.



# Запахи кода

Запахи плохого кода (от английского *Code Smells*) – это ключевые признаки необходимости рефакторинга. В процессе рефакторинга мы избавляемся от запахов, обеспечивая возможность дальнейшего развития приложения с той же или большей скоростью.



Отсутствие регулярного рефакторинга с течением времени способно полностью парализовать проект. И тогда вам, скорее всего, придётся выбросить в мусорник результат нескольких лет разработки, а также потратить ещё парочку лет, чтобы переписать всё с нуля.

Поэтому убирайте запахи, пока они маленькие!



# Раздувальщики

Раздувальщики представляют код, методы и классы, которые раздулись до таких больших размеров, что с ними стало невозможно эффективно работать. Все эти запахи зачастую не появляются сразу, а нарастают в процессе эволюции программы (особенно когда никто не пытается бороться с ними).

## § Длинный метод

Метод содержит слишком большое число строк кода. Длина метода более десяти строк должна начинать вас беспокоить.

## § Большой класс

Класс содержит множество полей/методов/строк кода.

## § Одержанность элементарными типами

- Использование элементарных типов вместо маленьких объектов для небольших задач (например, валюта, диапазоны, специальные строки для телефонных номеров и т.п.)
- Использование констант для кодирования какой-то информации (например, константа `USER_ADMIN_ROLE = 1` для обозначения пользователей с ролью администратора).
- Использование строковых констант в качестве названий полей в массивах.

## § Длинный список параметров

Количество параметров метода больше трёх-четырёх.

## § Группы данных

Иногда в разных частях кода встречаются одинаковые группы переменных (например, параметры подключения к базе данных). Такие группы следует превращать в самостоятельные классы.

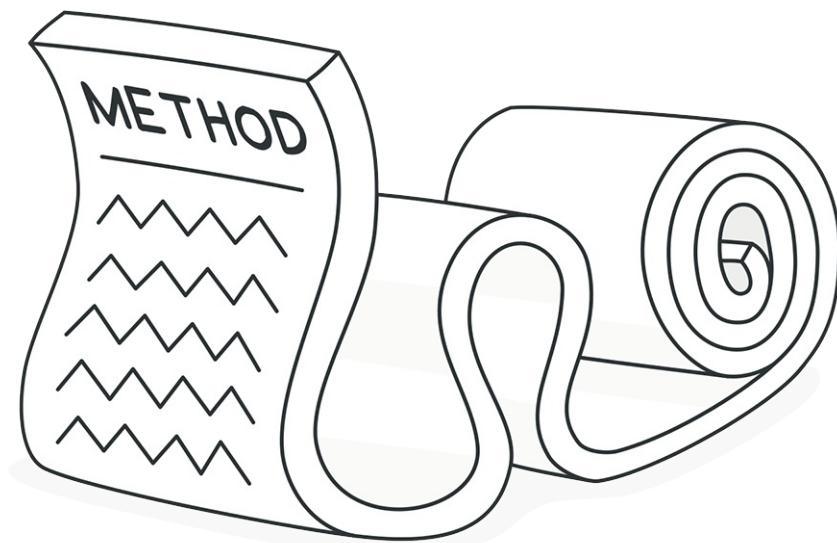


# Длинный метод

Также известен как *Long Method*

## Симптомы и признаки

Метод содержит слишком большое число строк кода. Длина метода более десяти строк должна начинать вас беспокоить.



## Причины появления

В метод всё время что-то добавляется, но ничего не выносится. Так как писать код намного проще, чем читать, этот запах долго остаётся незамеченным – до тех пор пока метод не превратится в настоящего монстра.

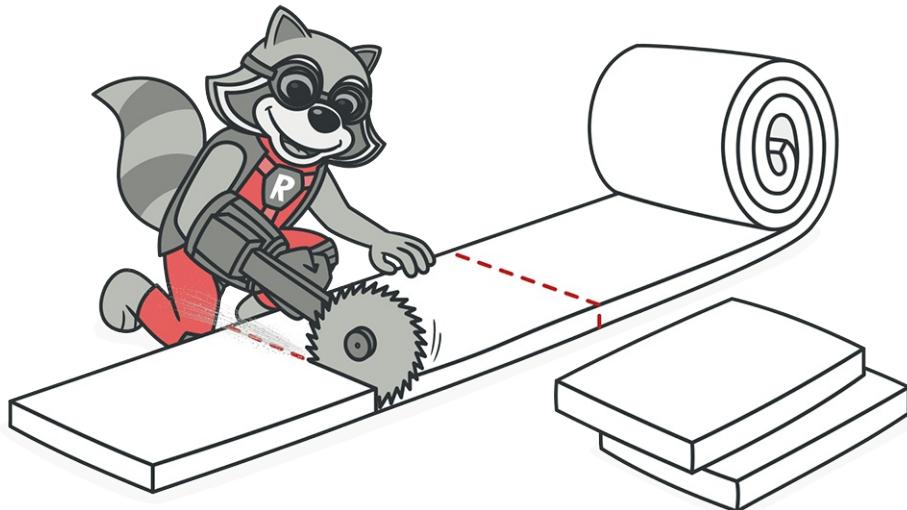
Стоит помнить, что человеку зачастую ментально сложнее создать новый метод, чем дописать что-то в уже существующий: «Как же, мне нужно добавить всего две строки, не буду же я создавать для этого целый метод».

Таким образом, добавляется одна строка за другой, а в результате метод превращается в большую тарелку спагетти.

## Лечение

Следует придерживаться такого правила: если ощущается необходимость что-то прокомментировать внутри метода, этот код лучше выделить в новый метод. Даже одну строку имеет смысл выделить в метод, если она нуждается в

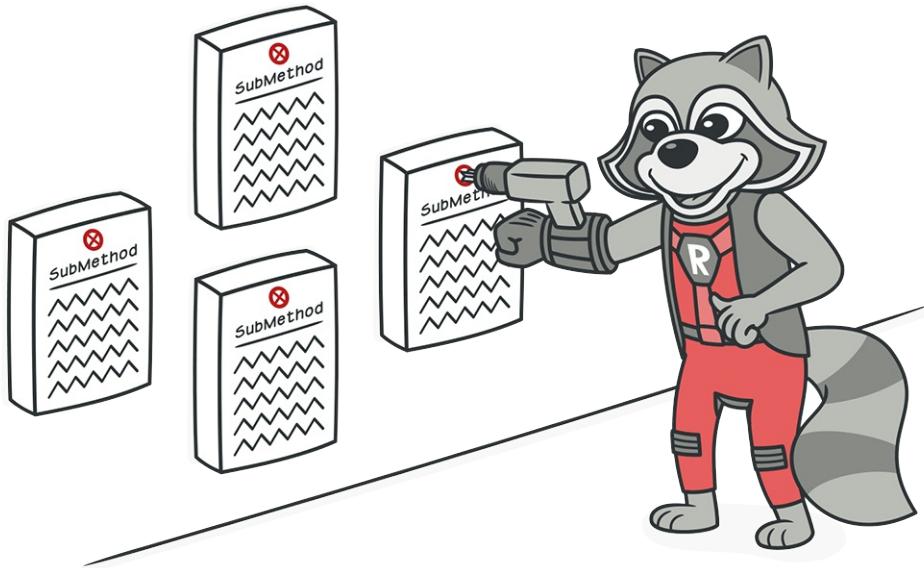
разъяснениях. К тому же, если у метода хорошее название, то не нужно будет смотреть в его код, чтобы понять, что он делает.



- Для сокращения тела метода достаточно применить извлечение метода.
- Если локальные переменные и параметры препятствуют выделению метода, можно применить замену временной переменной вызовом метода, замену параметров объектом и передачу всего объекта.
- Если предыдущие способы не помогли, можно попробовать выделить весь метод в отдельный объект с помощью замены метода объектом методов.
- Условные операторы и циклы свидетельствуют о возможности выделения кода в отдельный метод. Для работы с условными выражениями подходит декомпозиция условных операторов. Для работы с циклом – извлечение метода.

## Выигрыш

- Из всех видов объектного кода дольше всего выживают классы с короткими методами. Чем длиннее ваш метод или функция, тем труднее будет её понять и поддерживать.
- Кроме того, в длинных методах зачастую можно обнаружить «залежи» дублирования кода.



## Производительность

Многие волнуются, что увеличение числа методов может плохо сказаться на производительности. В абсолютном большинстве случаев, это не является реальной проблемой, так что **просто перестаньте об этом думать**.

Имея чистый и понятный код, вы с большей вероятностью натолкнётесь на отличный способ реструктуризировать код программы и увеличить реальную производительность, если такая надобность вообще будет.

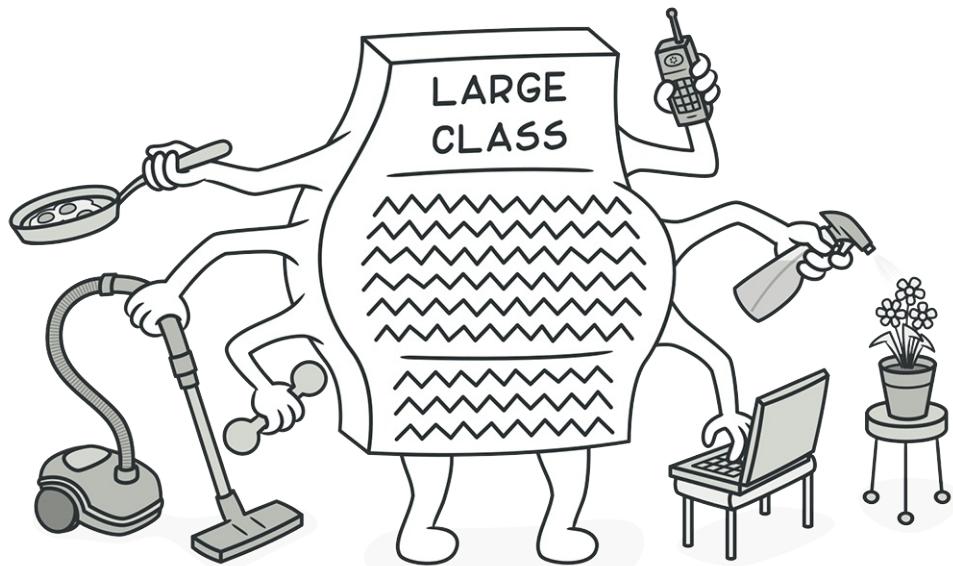


# Большой класс

Также известен как *Large Class*

## Симптомы и признаки

Класс содержит множество полей/методов/строк кода.



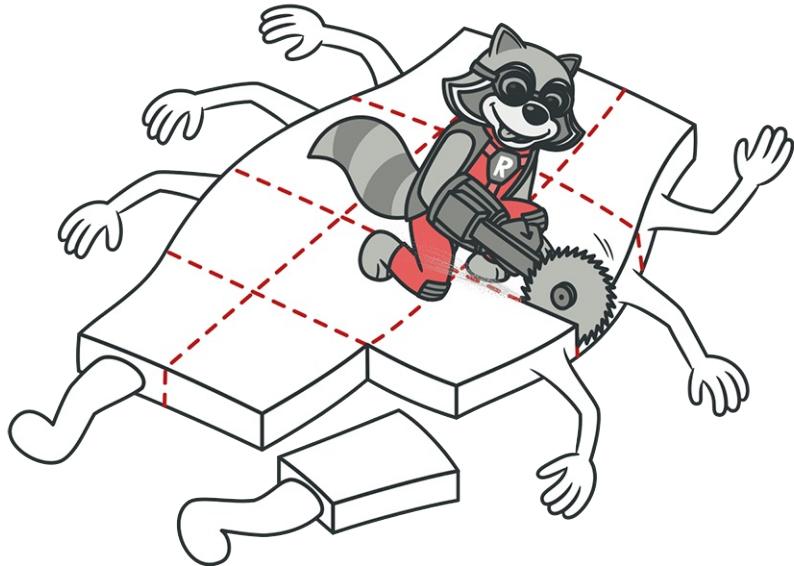
## Причины появления

Классы редко бывают большими изначально. Но со временем постепенно многие из них «раздуваются» в связи с развитием программы.

Как и в случае с длинными методами, чаще всего программисту ментально проще добавить фичу в существующий класс, чем создать новый класс для этой фичи.

## Лечение

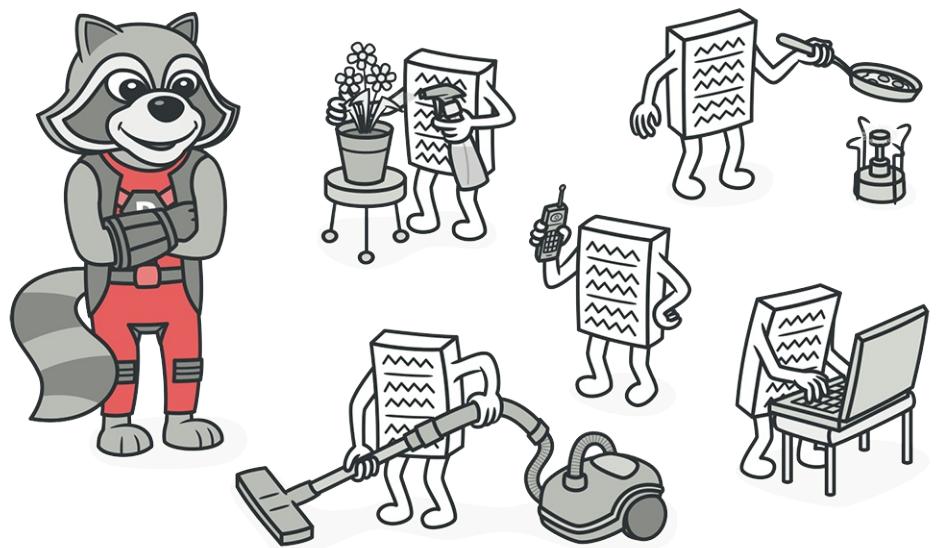
Когда класс реализует слишком обширный функционал, стоит подумать о его разделении:



- **Извлечение класса** поможет, если часть **поведения большого класса** может быть выделена в свой собственный компонент.
- **Извлечение подкласса** поможет, если часть **поведения большого класса** может иметь альтернативные реализации либо используется в редких случаях.
- **Извлечение интерфейса** поможет, если нужно иметь **список операций и поведений**, которые клиент сможет использовать.
- В классах графического интерфейса часто можно найти данные и поведения, которые не относятся к непосредственной отрисовке интерфейса, а скорее отвечают за общую логику работы. Такие данные и поведения следует выделить в отдельный класс предметной области, который бы управлял работой графического интерфейса. При этом может оказаться необходимым хранить копии некоторых данных в двух местах и обеспечить их согласованность. **Дублирование видимых данных** предлагает путь, которым можно это осуществить.

## Выигрыш

- Рефакторинг таких классов избавит разработчиков от необходимости запоминать чрезмерное количество имеющихся у класса атрибутов.
- Во многих случаях разделение больших классов на части позволяет избежать дублирования кода и функциональности.



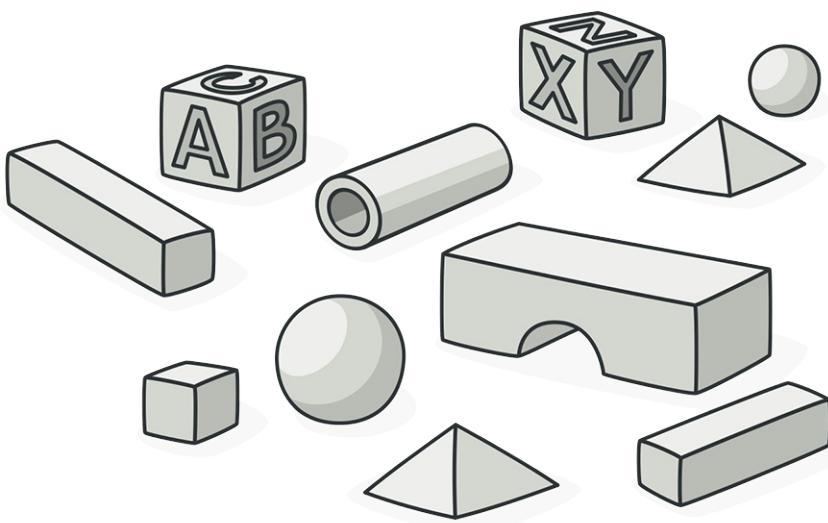


# Одержанность элементарными типами

Также известен как *Primitive Obsession*

## Симптомы и признаки

- Использование элементарных типов вместо маленьких объектов для небольших задач (например, валюта, диапазоны, специальные строки для телефонных номеров и т.п.)
- Использование констант для кодирования какой-то информации (например, константа `USER_ADMIN_ROLE = 1` для обозначения пользователей с ролью администратора).
- Использование строковых констант в качестве названий полей в массивах.



## Причины появления

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.

## Лечение

- 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**.



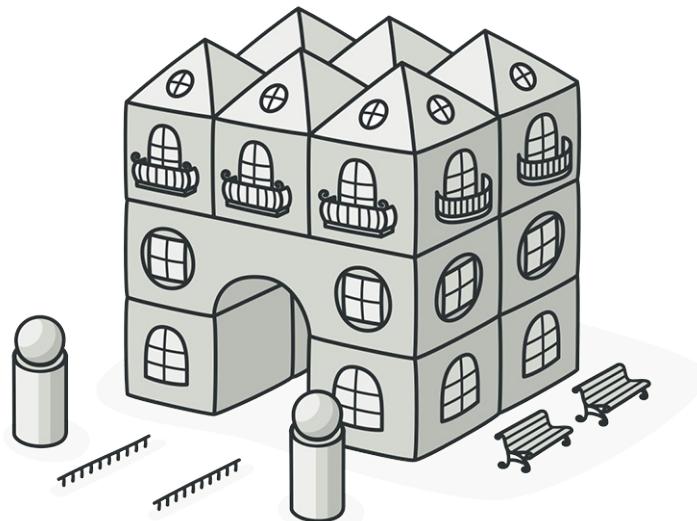
- Если значения этих примитивных полей используются в параметрах методов, используйте **замену параметров объектом** или **передачу всего объекта**.
- В случаях, когда в переменных закодированы какие-то сложные данные, используйте **замену кодирования типа классом**, **замену кодирования типа подклассами** или **замену кодирования типа состоянием/стратегией**.
- Если среди переменных есть массивы, используйте **замену массива объектом**.

## Выигрыш

- Повышает гибкость кода ввиду использования объектов вместо примитивных типов.
- Улучшает понимание и организацию кода. Операции над определёнными данными теперь собраны в одном месте, и их не надо искать по всему коду.

Теперь не нужно догадываться, зачем созданы все эти странные константы и почему поля содержатся в массиве.

- Может вскрыть факты дублирования кода.



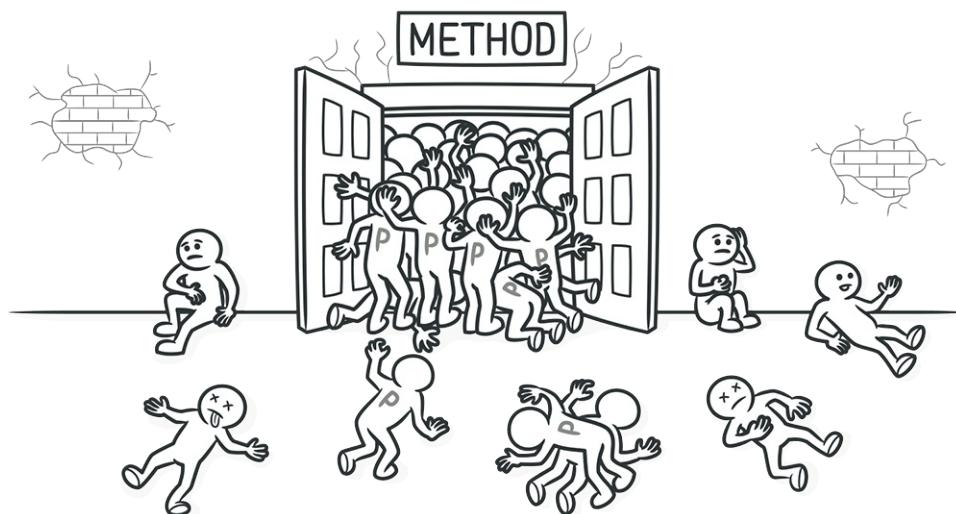


# Длинный список параметров

Также известен как *Long Parameter List*

## Симптомы и признаки

Количество параметров метода больше трёх-четырёх.



## Причины появления

Длинный список параметров может появиться после объединения нескольких вариантов алгоритмов в одном методе. В этом случае может быть создан длинный список параметров, контролирующих то, какая из вариаций будет выполнена и как.

Появление длинного списка параметров также может быть связано с попыткой программиста уменьшить связность между классами. Например, код создания конкретных объектов, нужных в методе, переместили из самого метода в код вызова этого метода, причём созданные объекты передаются в метод как параметры. Таким образом, оригинальный класс перестал знать о связях между объектами, и связность уменьшилась.

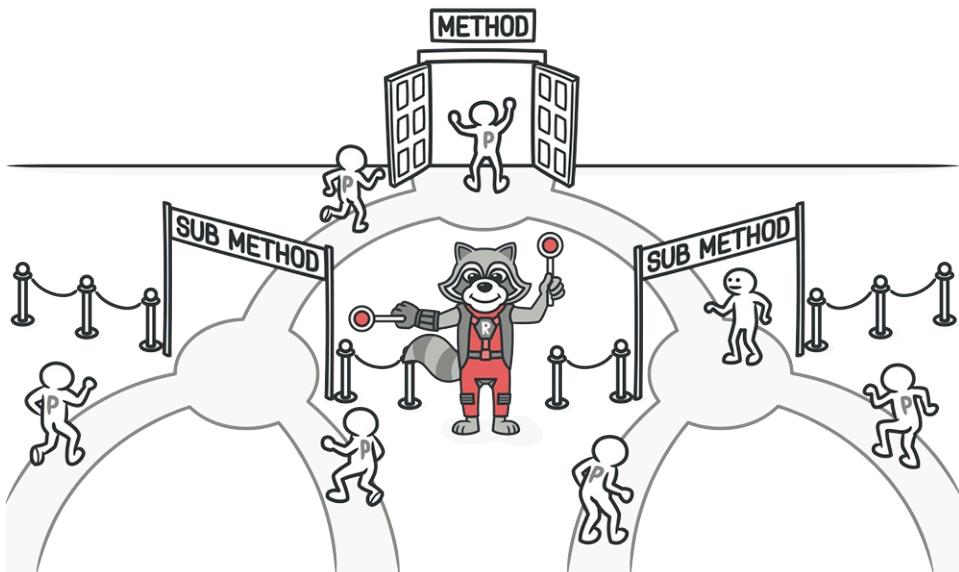
Но если таких объектов нужно создать несколько, под каждый из них потребуется свой параметр, что приводит к разрастанию списка параметров.

В длинных списках параметров трудно разбираться, они становятся противоречивыми и сложными в использовании. Вместо длинного списка параметров метод может использовать данные своего собственного объекта. Если

всех необходимых данных в текущем объекте нет, в качестве параметра метода можно передать другой объект, который получит недостающие данные.

## Лечение

- Если данные, передаваемые в метод, можно получить путём вызова метода другого объекта, применяем замену параметра вызовом метода. Этот объект может быть помещён в поле собственного класса либо передан как параметр метода.



- Вместо того чтобы передавать группу данных, полученных из другого объекта в качестве параметров, в метод можно передать сам объект, используя передачу всего объекта.
- Если есть несколько несвязанных элементов данных, иногда их можно объединить в один объект-параметр, применив замену параметров объектом.

## Выигрыш

- Повышает читабельность кода, уменьшает его размер.
- В процессе рефакторинга вы можете обнаружить дублирование кода, которое ранее было незаметно.

## Не стоит трогать, если...

- Не стоит избавляться от параметров, если при этом появляется нежелательная связанность между классами.

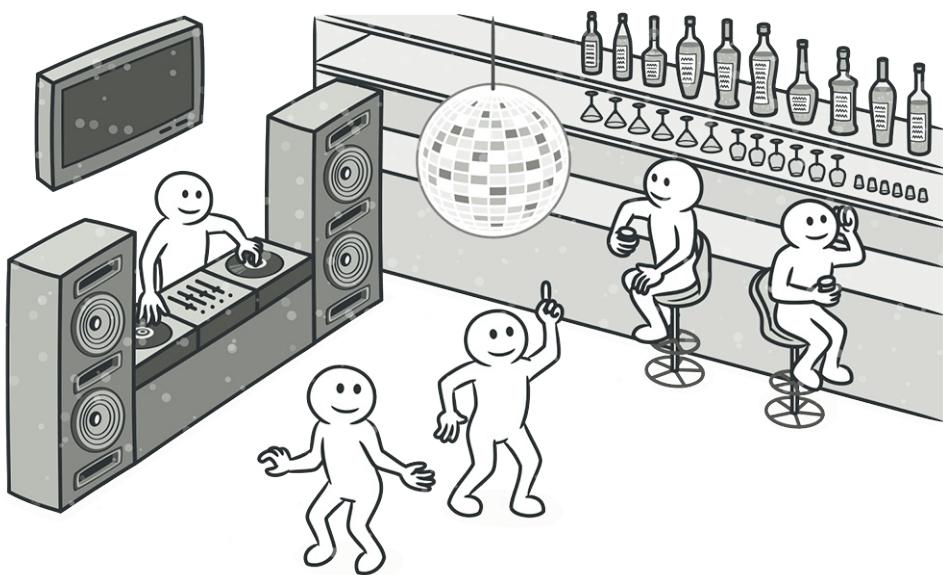


# Группы данных

Также известен как *Data Clumps*

## Симптомы и признаки

Иногда в разных частях кода встречаются одинаковые группы переменных (например, параметры подключения к базе данных). Такие группы следует превращать в самостоятельные классы.



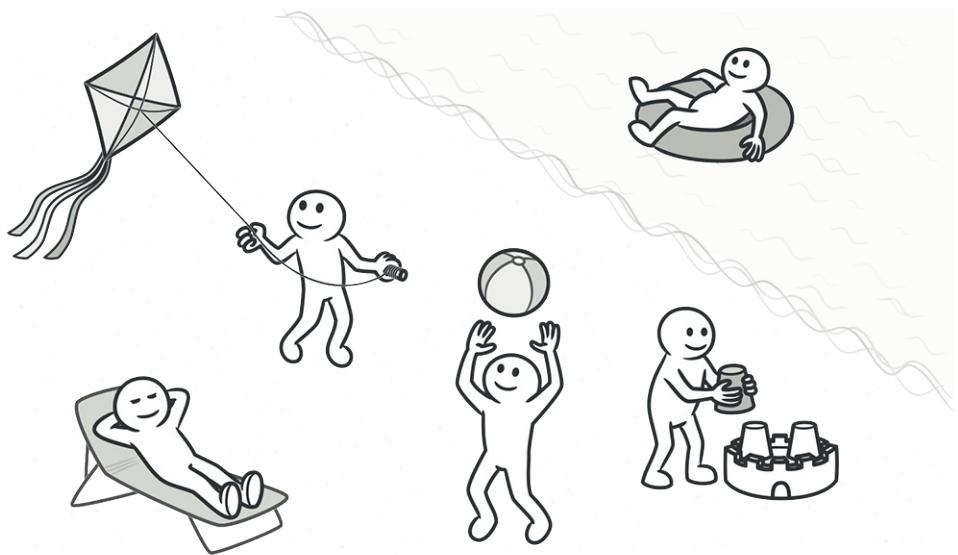
## Причины появления

Появление групп данных является следствием плохой структурированности программы или программирования методом копирования-вставки.

Чтобы определить группу данных, достаточно удалить одно из значений данных и проверить, сохранят ли смысл остальные. Если нет, это верный признак того, что группа переменных напрашивается на объединение их в объект.

## Лечение

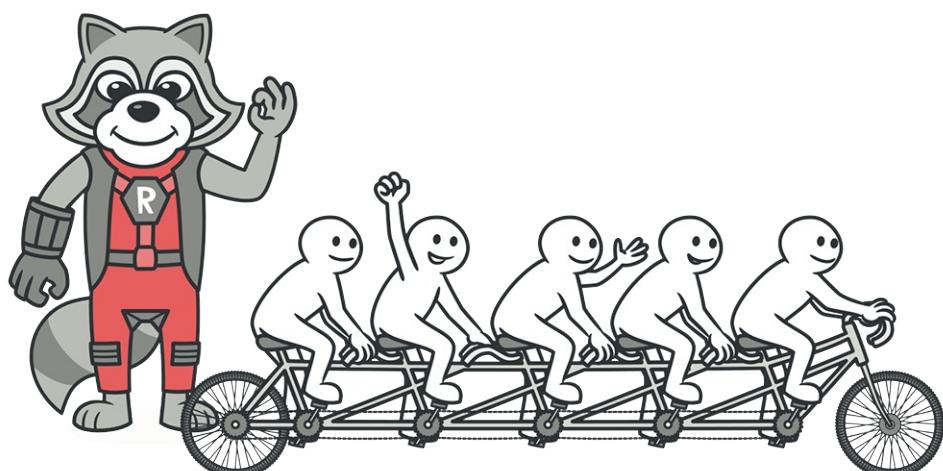
- Если повторяющиеся данные являются полями какого-то класса, используйте извлечение класса для перемещения полей в собственный класс.



- Если те же группы данных передаются в параметрах методов, используйте **замену параметров объектом** чтобы выделить их в общий класс.
- Если некоторые из этих данных передаются в другие методы, подумайте о возможности передачи в метод всего объекта данных вместо отдельных полей (в этом поможет **передача всего объекта**).
- Посмотрите на код, который использует эти поля. Возможно, имеет смысл перенести этот код в класс данных.

## Выигрыш

- Улучшает понимание и организацию кода. Операции над определёнными данными теперь собраны в одном месте, и их не надо искать по всему коду.
- Уменьшает размер кода.



## **Не стоит трогать, если...**

- Передача всего объекта в параметрах метода вместо передачи его значений (элементарных типов) может создать нежелательную зависимость между двумя классами.



# Нарушители объектно-ориентированного дизайна

Все эти запахи являются собой неполное или неправильное использование возможностей объектно-ориентированного программирования.

## § Операторы switch

У вас есть сложный оператор `switch` или последовательность `if`-ов.

## § 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.

## § 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.

## § Альтернативные классы с разными интерфейсами

Два класса выполняют одинаковые функции, но имеют разные названия методов.

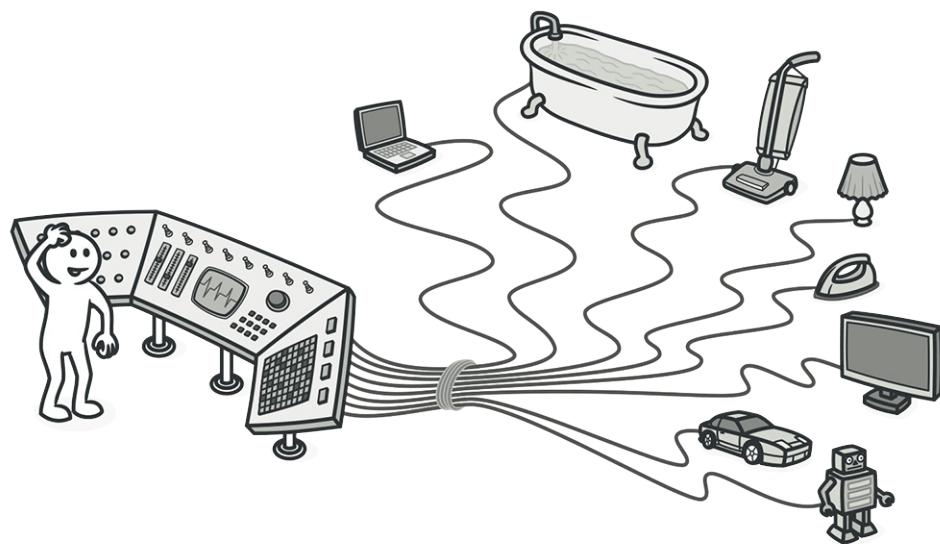


# ⚡ Операторы switch

Также известен как *Switch Statements*

## Симптомы и признаки

У вас есть сложный оператор `switch` или последовательность `if`-ов.



## Причины появления

Одним из очевидных признаков объектно-ориентированного кода служит сравнительно редкое использование операторов типа `switch` или `case`. Часто один и тот же блок `switch` оказывается разбросанным по разным местам программы. При добавлении в него нового варианта приходится искать все эти блоки `switch` и модифицировать их.

Как правило, заметив блок `switch`, следует подумать о полиморфизме.

## Лечение

- Чтобы изолировать `switch` и поместить его в нужный класс может понадобиться извлечение метода и перемещение метода.
- Если `switch` переключается по коду типа, например, переключается режим выполнения программы, то следует использовать замену кодирования типа подклассами или замену кодирования типа состоянием/стратегией.

- После настройки структуры наследования следует использовать замену условного оператора полиморфизмом.
- Если вариантов в операторе не очень много и все они приводят к вызову одного и того же метода с разными параметрами, введение полиморфизма будет избыточным. В этом случае стоит задуматься о разбиении этого метода на несколько разных, которые будут выполнять каждый свои функции, для чего нужно применить замену параметра набором специализированных методов.
- Если одним из вариантов условного оператора является `null`, используйте введение Null-объекта.

## Выигрыш

- Улучшает организацию кода.



## Не стоит трогать, если...

- Когда оператор `switch` выполняет простые действия, нет никакого смысла что-то менять в коде.
- Зачастую оператор `switch` используется фабричных паттернах проектирования (Фабричный метод, Абстрактная фабрика), для выбора создаваемого класса.

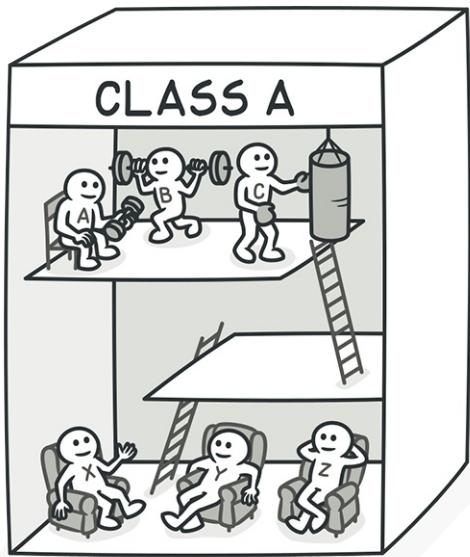


# Временное поле

Также известен как *Temporary Field*

## Симптомы и признаки

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



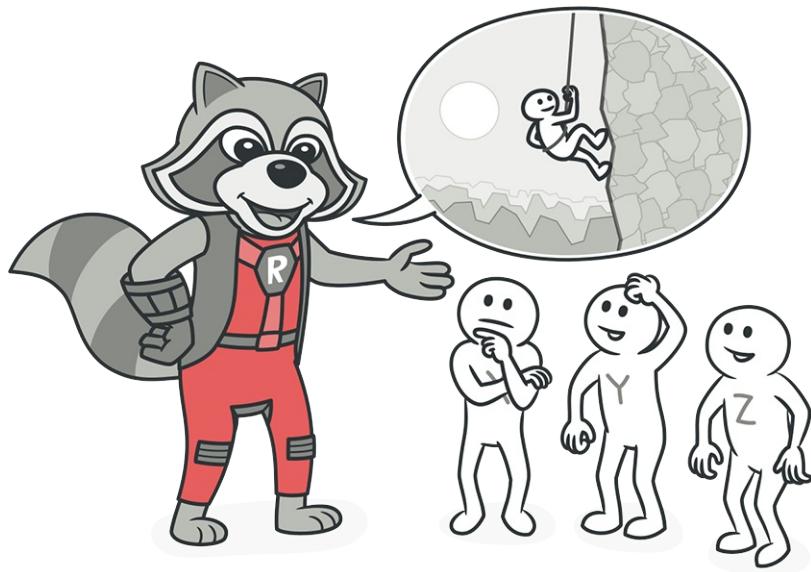
## Причины появления

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 пропадают.

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

## Лечение

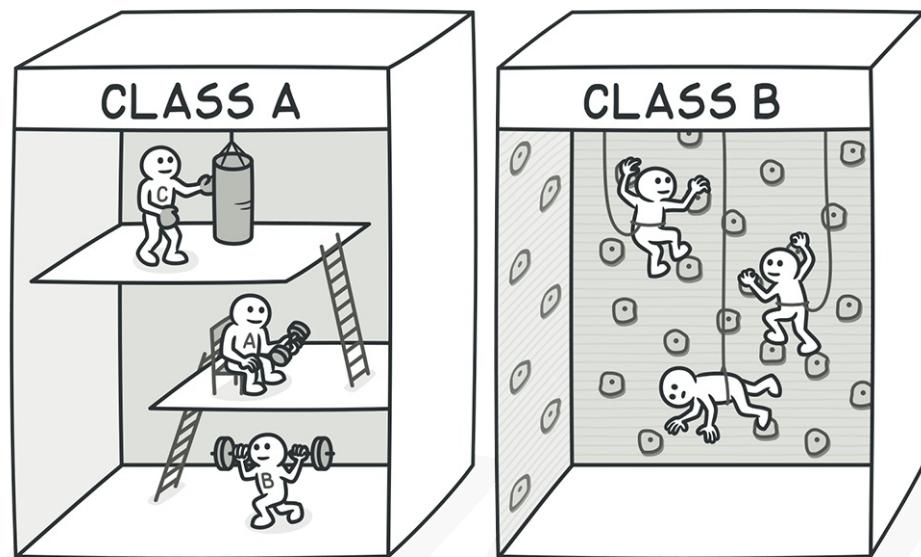
- Временные поля и весь код, работающий с ними, можно поместить в свой собственный класс с помощью извлечения класса. По сути, вы создадите объект-метод.



- Введите **Null-объект** и встройте его вместо кода проверок на наличие значений во временных полях.

## Выигрыш

- Улучшает понятность и организацию кода.



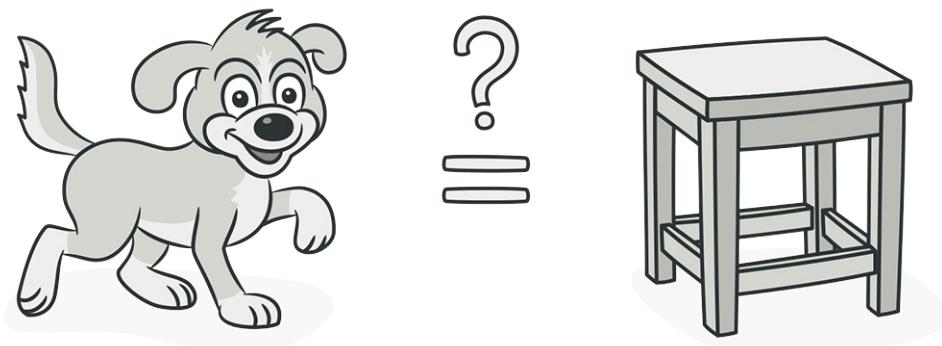


# Отказ от наследства

Также известен как *Refused Bequest*

## Симптомы и признаки

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.

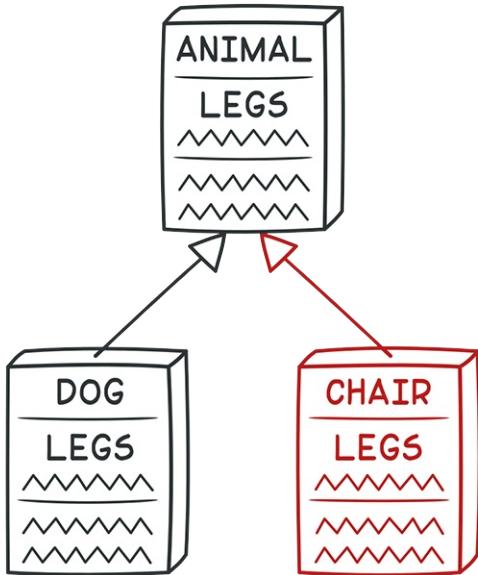


## Причины появления

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.

## Лечение

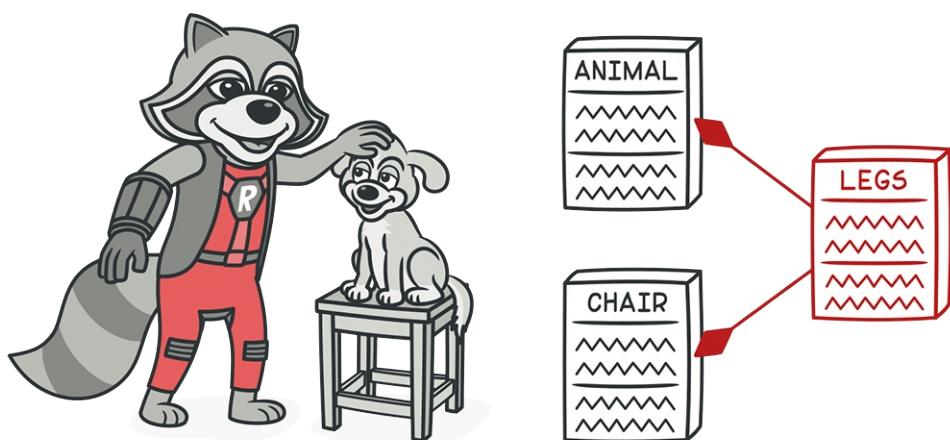
- Если наследование не имеет смысла, и подкласс в действительности не является представителем суперкласса, следует избавиться от отношения наследования между этими классами, применив замену наследования делегированием.



- 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).

## Выигрыш

- Улучшает понимание и организацию кода. Теперь вы не будете тратить время на догадки о том, почему класс Стул унаследован от класса Животное (несмотря на то, что оба имеют четыре ноги).





# Альтернативные классы с разными интерфейсами

Также известен как *Alternative Classes with Different Interfaces*

## Симптомы и признаки

Два класса выполняют одинаковые функции, но имеют разные названия методов.



## Причины появления

Программист, который создал один из классов, скорей всего, не знал о том, что в программе уже существует аналогичный по функциям класс.

## Лечение

Постарайтесь привести интерфейс классов к общему знаменателю:

- Переименуйте методы так, чтобы они стали одинаковыми во всех альтернативных классах.
- Используйте перемещение метода, добавление параметра и параметризацию метода для того, чтобы сигнатура и реализация методов стали одинаковыми.
- Если только часть функциональности классов идентична, попробуйте извлечь эту часть в общий суперкласс. Существующие классы в этом случае станут подклассами.
- После того, как вы определились с вариантом «лечения» и осуществили его, подумайте, возможно, один из классов теперь можно удалить.

## Выигрыш

- Вы избавляетесь от ненужного дублирования кода, и, таким образом, уменьшаете его размер.
- Повышается читабельность кода, улучшается его понимание. Вам больше не придется гадать, зачем создавался второй класс, выполняющий точно такие же функции, как и первый.



## Не стоит трогать, если...

- Иногда объединить классы оказывается невозможno либо настолько сложно, что смысла заниматься этой работой нет. Один из примеров – альтернативные классы находятся в двух разных библиотеках, каждая из которых имеет свою версию класса.



# Утяжелители изменений

Эти запахи приводят к тому, что при необходимости что-то поменять в одном месте программы, вам приходится вносить множество изменений в других местах. Это серьезно осложняет и удорожает развитие программы.

## § Расходящиеся модификации

При внесении изменений в класс приходится изменять большое число различных методов. Например, для добавления нового вида товара вам нужно изменить методы поиска, отображения и заказа товаров.

## § Стрельба дробью

При выполнении любых модификаций приходится вносить множество мелких изменений в большое число классов.

## § Параллельные иерархии наследования

Всякий раз при создании подкласса какого-то класса приходится создавать ещё один подкласс для другого класса.



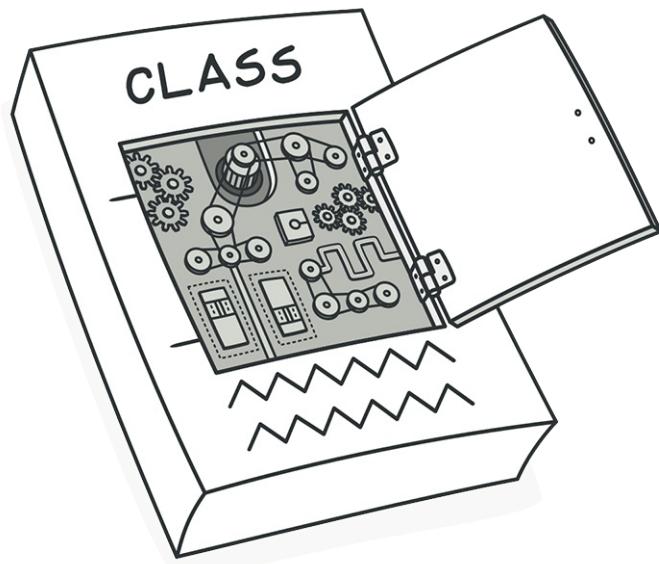
# Расходящиеся модификации

Также известен как *Divergent Change*

«Расходящиеся модификации» похожи на [Стрельбу дробью](#), но на самом деле являются ее противоположностью. «Расходящиеся модификации» имеют место, когда есть один класс, в котором производится много разных изменений, а «Стрельба дробью» – это одно изменение, затрагивающее одновременно много классов.

## Симптомы и признаки

При внесении изменений в класс приходится изменять большое число различных методов. Например, для добавления нового вида товара вам нужно изменить методы поиска, отображения и заказа товаров.



## Причины появления

Часто появление расходящихся модификаций является следствием плохой структурированности программы или программирования методом копирования-вставки.

## Лечение

- Разделите поведения класса, используя извлечение класса.
- Если различные классы имеют одно и то же поведение, возможно, следует объединить эти классы, используя наследование (извлечение суперкласса и извлечение подкласса).

## Выигрыш

- Улучшает организацию кода.
- Уменьшает дублирование кода.
- Упрощает поддержку.





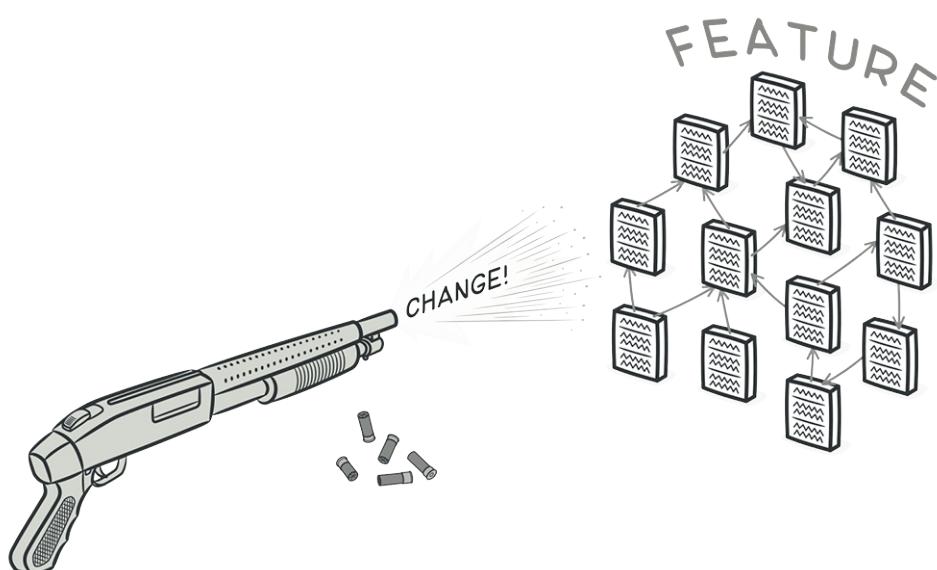
# Стрельба дробью

Также известен как *Shotgun Surgery*

«Стрельба дробью» похожа на Расходящиеся модификации, но является противоположностью этого запаха. «Расходящиеся модификации» имеют место, когда есть один класс, в котором производится много различных изменений, а «Стрельба дробью» – это одно изменение, затрагивающее много классов.

## Симптомы и признаки

При выполнении любых модификаций приходится вносить множество мелких изменений в большое число классов.



## Причины появления

Одна обязанность была разделена среди множества классов. Это может случиться после фанатичного исправления Расходящихся модификаций.

## Лечение

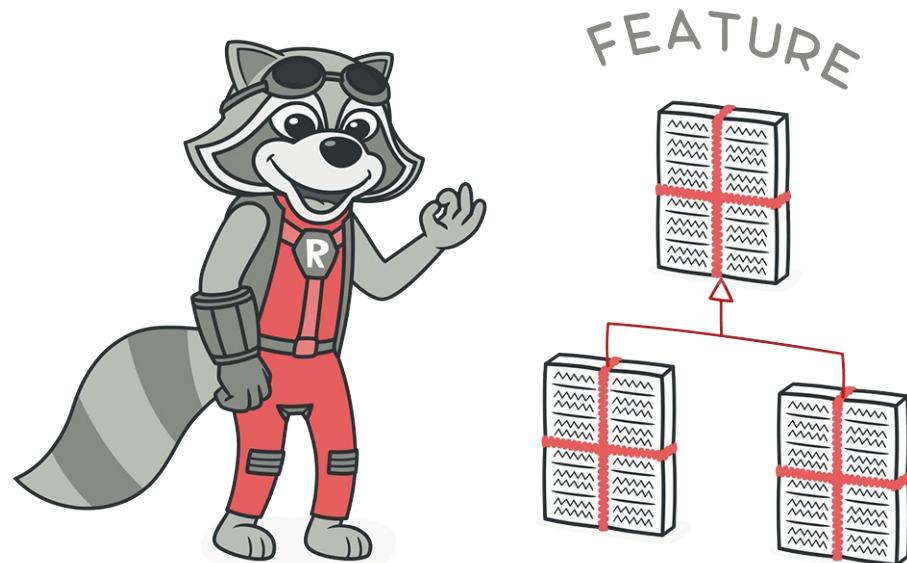
- Вынести все изменения в один класс позволяют перемещение метода и перемещение поля. Если для выполнения этого действия нет подходящего класса, то следует предварительно создать новый.



- Если после вынесения кода в один класс в оригинальных классах мало что осталось, следует попытаться от них избавиться, воспользовавшись встраиванием класса.

## Выигрыш

- Улучшает организацию кода.
- Уменьшает дублирование кода.
- Упрощает поддержку.



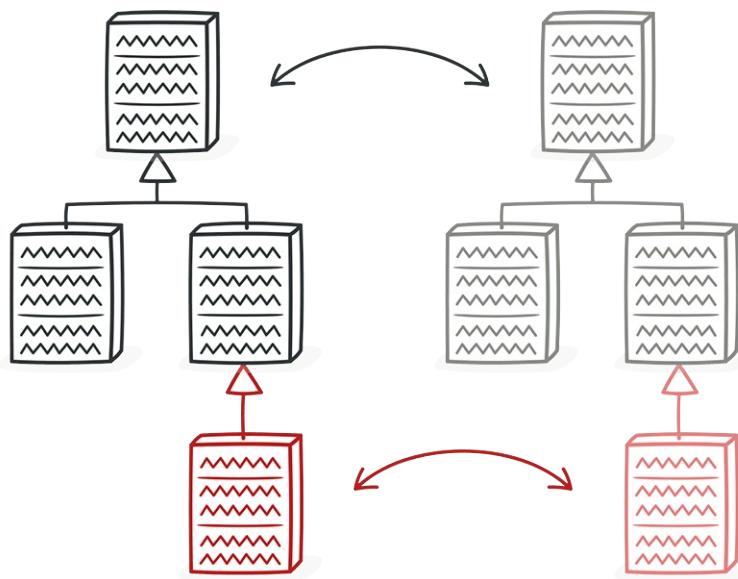


# Параллельные иерархии наследования

Также известен как *Parallel Inheritance Hierarchies*

## Симптомы и признаки

Всякий раз при создании подкласса какого-то класса приходится создавать ещё один подкласс для другого класса.



## Причины появления

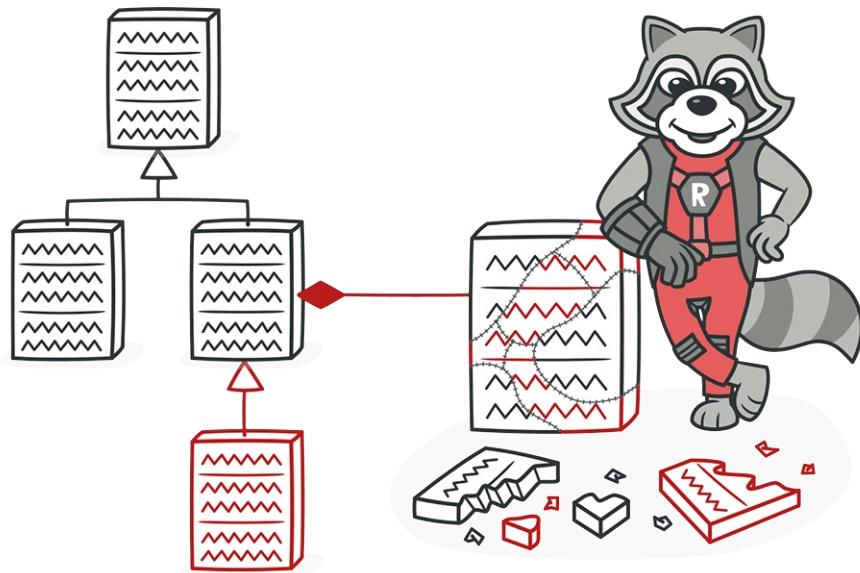
Пока иерархия была небольшая, всё было хорошо. Но с появлением новых классов вносить изменения становилось всё сложнее и сложнее.

## Лечение

- Вы можете попытаться устраниТЬ дублирования параллельных классов в два этапа. Во-первых, нужно заставить экземпляры одной иерархии ссылаться на экземпляры другой иерархии. Затем следует убрать иерархию в ссылающемся классе с помощью перемещения метода и перемещения поля.

## Выигрыш

- Уменьшает дублирования кода.
- Может улучшить организацию кода.



## Не стоит трогать, если...

- Иногда наличие паралельной иерархии – это необходимое зло, без которого устройство программы было бы еще хуже. Если вы обнаружите, что ваши попытки устраниить дублирование приводят к еще большему ухудшению организации кода, то... остановите рефакторинг, откатите все внесенные изменения, выпейте чаю и начните привыкать к этому коду.



# Замусориватели

Замусориватели являются собой что-то бесполезное и лишнее, от чего можно было бы избавиться, сделав код чище, эффективней и проще для понимания.

## § Комментарии

Метод содержит множество поясняющих комментариев.

## § Дублирование кода

Два фрагмента кода выглядят почти одинаковыми.

## § 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.

## § Класс данных

Классы данных – это классы, которые содержат только поля и простейшие методы для доступа к ним (геттеры и сеттеры). Это просто контейнеры для данных, используемые другими классами. Эти классы не содержат никакой дополнительной функциональности и не могут самостоятельно работать с данными, которыми владеют.

## § Мёртвый код

Переменная, параметр, поле, метод или класс больше не используются (чаще всего потому, что устарели).

## § Теоретическая общность

Класс, метод, поле или параметр не используются.

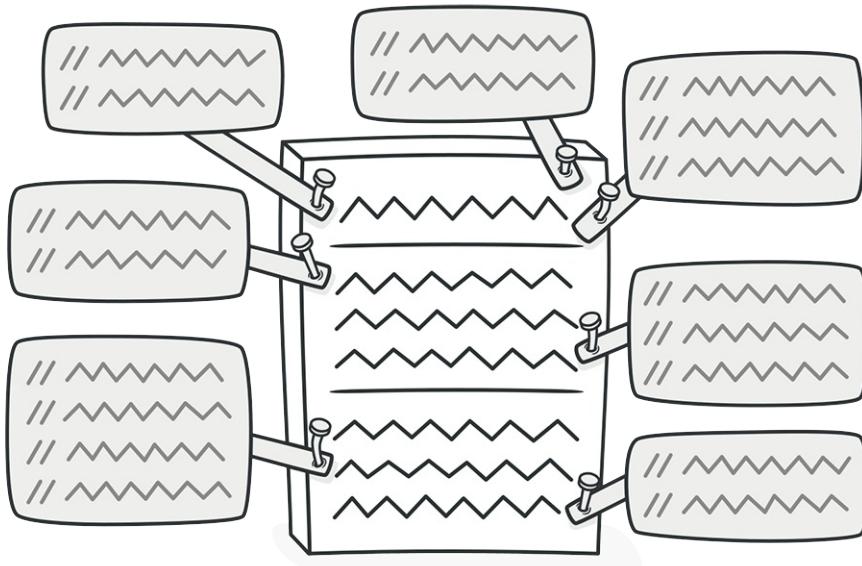


# Комментарии

Также известен как *Comments*

## Симптомы и признаки

Метод содержит множество поясняющих комментариев.



## Причины появления

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.

## Лечение

- Если комментарий предназначен для того, чтобы объяснить сложное выражение, возможно, это выражение лучше разбить на понятные подвыражения с помощью извлечения переменной.
- Если комментарий поясняет целый блок кода, возможно, этот блок можно извлечь в отдельный метод с помощью извлечения метода. 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.
- Если требуется описать какие-то правила, касающиеся правильной работы метода, попробуйте рефакторинг введение утверждения.

## Выигрыш

- Код становится более очевидным и понятным.



## Не стоит трогать, если...

Иногда комментарии бывают полезными:

- Те, которые объясняют **почему** что-то выполняется именно таким образом.
- Те, которые объясняют сложные алгоритмы (когда все иные средства упростить алгоритм уже были испробованы).

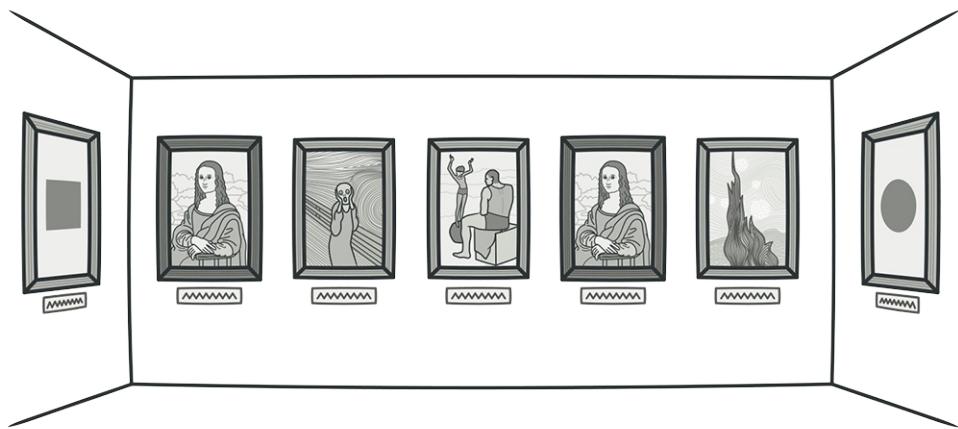


# Дублирование кода

Также известен как *Duplicate Code*

## Симптомы и признаки

Два фрагмента кода выглядят почти одинаковыми.



## Причины появления

В большинстве случаев дублирование возникает тогда, когда в проекте работает несколько человек, причём над разными его частями. Они работают над похожими задачами, но не знают, что коллега уже написал похожий код, который можно использовать вместо написания своего.

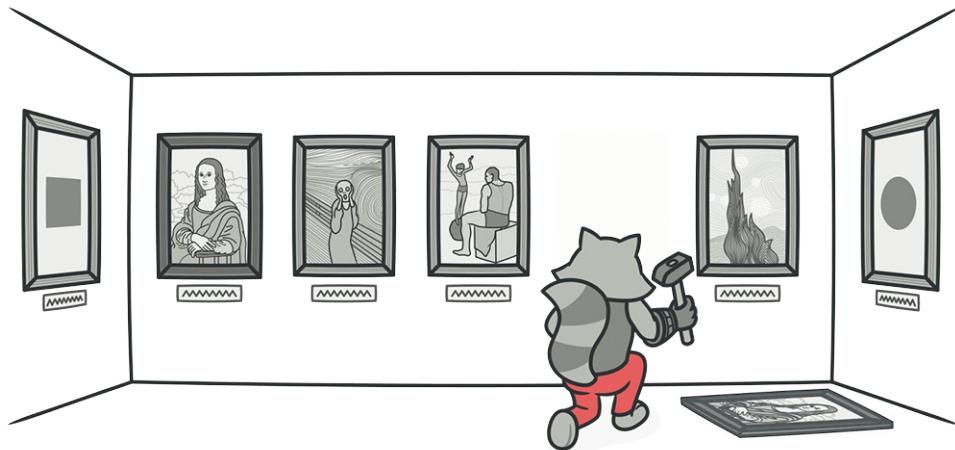
Встречается и косвенное дублирование, когда конкретные участки кода отличаются внешне, хотя и выполняют одну и ту же задачу. Такое дублирование бывает довольно сложно обнаружить и исправить.

В отдельных случаях дублирование создаётся намеренно. Зачастую, в спешке, когда поджимают сроки сдачи проекта. Начинающий программист видит в уже написанном коде фрагмент, выглядящий «почти так, как нужно» и не может устоять перед соблазном просто скопировать код куда-то в другое место (и так десяток раз).

А в самых запущенных случаях программист просто слишком ленив, чтобы избавить код от дублирования.

# Лечение

- Один и тот же участок кода присутствует в двух методах одного и того же класса: необходимо применить **извлечение метода** и вызывать код созданного метода из обоих участков.



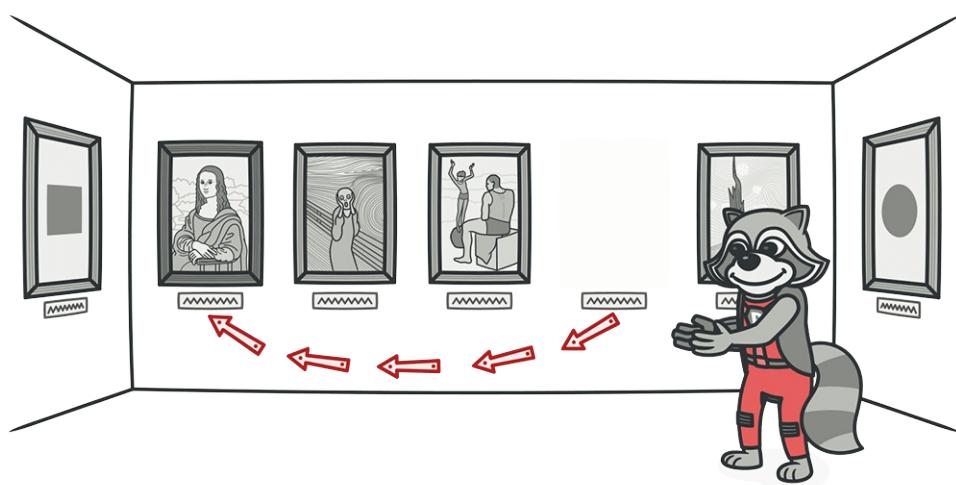
- Один и тот же участок кода присутствует в двух подклассах, находящихся на одном уровне:
  - Необходимо применить **извлечение метода** для обоих классов с последующим **подъёмом поля** для полей, которые используются в поднятом методе.
  - Если общий код находится в конструкторе, следует использовать **подъём тела конструктора**.
  - Если участки кода похожи, но не совпадают полностью, нужно пользоваться **созданием шаблонного метода**.
  - Если оба метода делают одно и то же, но с помощью разных алгоритмов, можно выбрать более чёткий из этих алгоритмов и применить **замещение алгоритма**.
- Дублирующийся код находится в двух разных классах:
  - Если эти классы не являются частью какой-то иерархии, следует использовать **извлечение суперкласса**, чтобы создать для интересующих классов один суперкласс, содержащий всю общую функциональность.
  - Если создание суперкласса нежелательно или невозможно, следует применить **извлечение класса** в одном классе, а затем использовать новый

компонент в другом.

- Присутствует череда условных операторов, которые исполняют один и тот же код и отличаются только условиями, следует объединить эти операторы в один с общим условием с помощью **объединения условных операторов**, а также применить **извлечение метода**, чтобы вынести это **условие в отдельный метод с понятным названием**.
- Один и тот же код выполняется во всех ветках условного оператора: необходимо вынести одинаковый код за пределы условного оператора с помощью **объединения дублирующихся фрагментов в условных операторах**.

## Выигрыш

- Объединение дублирующего кода позволяет улучшить структуру кода и уменьшить его объём.
- Это, в свою очередь, ведёт к упрощению и удешевлению поддержки кода в будущем.



## Не стоит трогать, если...

- В очень редких случаях объединение двух одинаковых участков кода может сделать код менее очевидным и понятным.



# Ленивый класс

Также известен как *Lazy Class*

## Симптомы и признаки

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.



## Причины появления

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.

## Лечение

- Почти бесполезные компоненты должны быть подвергнуты встраиванию класса.



- При наличии подклассов с недостаточными функциями попробуйте свёртывание иерархии.

## Выигрыш

- Уменьшение размера кода.
- Упрощение поддержки.

## Не стоит трогать, если...

- Иногда *Ленивый класс* бывает создан для того, чтобы явно очертировать какие-то намерения. В этом случае, стоит соблюдать баланс понятности кода и его простоты.

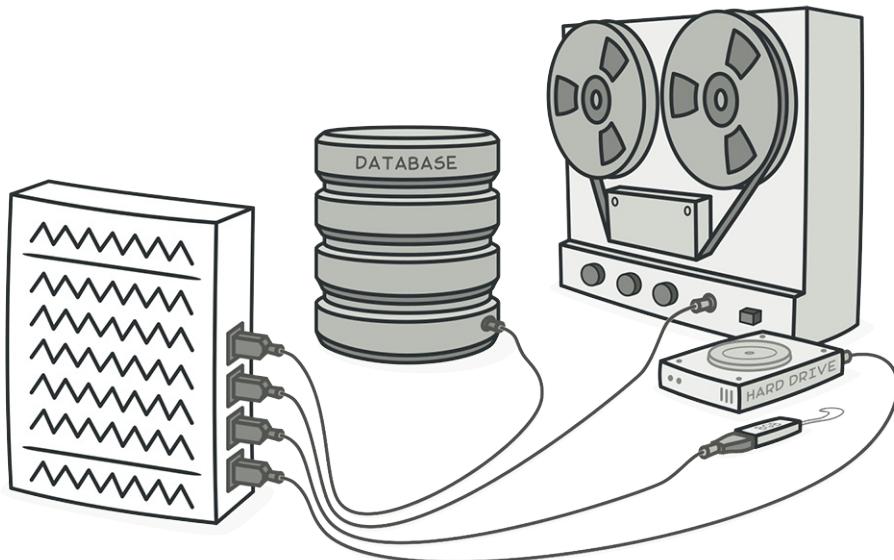


# Класс данных

Также известен как *Data Class*

## Симптомы и признаки

Классы данных – это классы, которые содержат только поля и простейшие методы для доступа к ним (геттеры и сеттеры). Это просто контейнеры для данных, используемые другими классами. Эти классы не содержат никакой дополнительной функциональности и не могут самостоятельно работать с данными, которыми владеют.



## Причины появления

Это нормально, когда класс в начале своей жизни содержит всего лишь несколько публичных полей (а может даже и парочку геттеров/сеттеров). Тем не менее, настоящая сила объектов заключается в том, что они могут хранить типы поведения или операции над собственными данными.

## Лечение

- Если класс содержит публичные поля, примените инкапсуляцию поля чтобы скрыть их из прямого доступа, разрешив доступ только через геттеры и сеттеры.

- Примените инкапсуляцию коллекции для данных, которые хранятся в коллекциях (вроде массивов).
- Осмотрите клиентский код, который использует этот класс. Возможно, там вы найдёте функциональность, которая смотрелась бы уместнее в самом классе данных. В этом случае используйте перемещение метода и извлечение метода для переноса функциональности в класс данных.



- После того, как класс наполнился осмысленными методами, возможно, стоит подумать об уничтожении старых методов доступа к данным, которые дают слишком открытый доступ к данным класса. В этом вам поможет удаление сеттера и сокрытие метода.

## Выигрыш

- Улучшает понимание и организацию кода. Операции над определёнными данными теперь собраны в одном месте, их не надо искать по всему коду.
- Может вскрыть факты дублирования клиентского кода.



# Мёртвый код

Также известен как *Dead Code*

## Симптомы и признаки

Переменная, параметр, поле, метод или класс больше не используются (чаще всего потому, что устарели).



## Причины появления

Когда требования к программному продукту изменились, либо были внесены какие-то корректировки, но чистка старого кода так и не была проведена.

Мёртвый код можно обнаружить и в сложном условном коде, где одна из веток никогда не может быть исполнена (в виду ошибки или другого стечения обстоятельств).

## Лечение

Лучше всего мёртвый код обнаруживается при помощи хорошей среды разработки (IDE).

- Удалите неиспользуемый код и лишние файлы.



- В случае выявления ненужного класса может быть использовано **встраивание класса**. Если у такого класса есть подклассы, то поможет **схлопывание иерархии**.
- Для удаления ненужных параметров используйте **удаление параметра**.

## Выигрыш

- Уменьшает размер кода.
- Упрощает его поддержку.

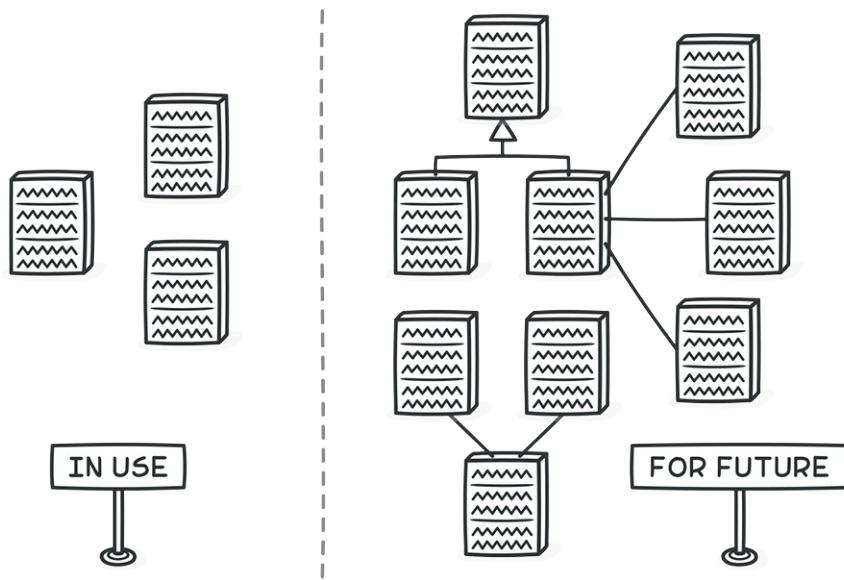


# Теоретическая общность

Также известен как *Speculative Generality*

## Симптомы и признаки

Класс, метод, поле или параметр не используются.



## Причины появления

Иногда код создаётся «про запас», чтобы поддерживать какой-то возможный будущий функционал, который в итоге так и не реализуется. В результате этот код становится труднее понимать и сопровождать.

## Лечение

- Для удаления незадействованных абстрактных классов используйте сворачивание иерархии.



- Ненужное делегирование функциональности другому классу может быть удалено с помощью встраивания класса.
- От неиспользуемых методов можно избавиться с помощью встраивания метода.
- Методы с неиспользуемыми параметрами должны быть подвергнуты удалению параметров.
- Неиспользуемые поля можно просто удалить.

## Выигрыш

- Уменьшение размера кода.
- Упрощение поддержки.

## Не стоит трогать, если...

- В случаях, когда вы работаете над фреймворком, создание функциональности, не используемой самим фреймворком, вполне оправдано. Главное, чтобы она была полезна пользователям фреймворка.
- Перед удалением элементов, стоит удостовериться, не используются ли они в юнит-тестах. Такое бывает, если в тестах необходим способ получения какой-то служебной информации класса или осуществления каких-то специальных тестовых действий.



# Опутыватели связями

Все запахи из этой группы приводят к избыточной связанности между классами, либо показывают, что бывает если тесная связанность заменяется постоянным делегированием.

## § Завистливые функции

Метод обращается к данным другого объекта чаще, чем к собственным данным.

## § Неуместная близость

Один класс использует служебные поля и методы другого класса.

## § Цепочка вызовов

Вы видите в коде цепочки вызовов вроде такой `$a->b()->c()->d()`

## § Посредник

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

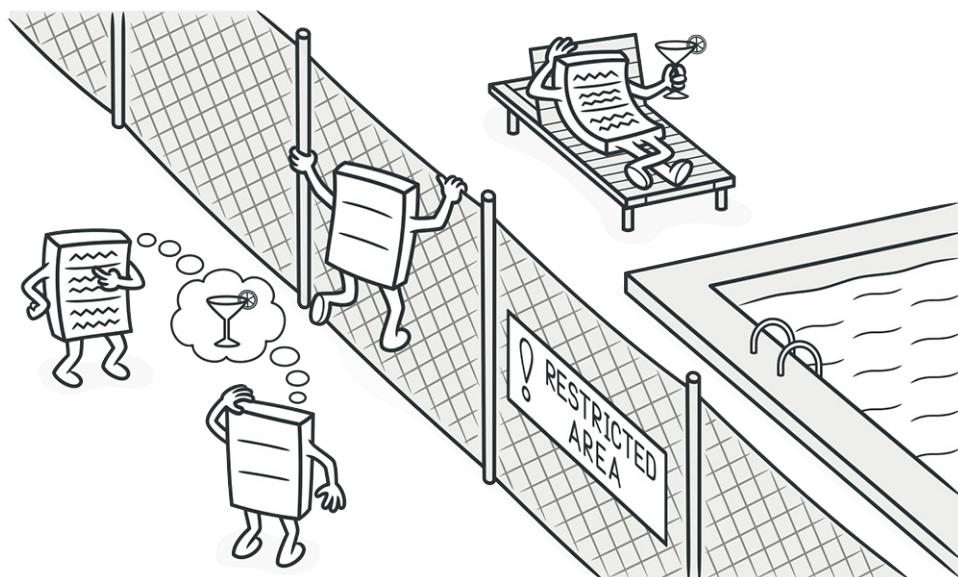


# ⌚ Завистливые функции

Также известен как *Feature Envy*

## Симптомы и признаки

Метод обращается к данным другого объекта чаще, чем к собственным данным.



## Причины появления

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.

## Лечение

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**.
- Если только часть метода обращается к данным другого объекта, примените **извлечение метода** к этой части.
- Если метод использует функции нескольких других классов, нужно сначала определить, в каком классе находится больше всего используемых данных. Затем следует переместить метод в этот класс вместе с остальными данными. Как альтернатива, с помощью **извлечения метода** метод разбивается на несколько частей, и они помещаются в разные места в других классах.

## Выигрыш

- Уменьшение дублирования кода (если код работы с данными переехал в одно общее место).
- Улучшение организации кода (т.к. методы работы с данными находятся возле этих данных).



## Не стоит трогать, если...

- Бывают случаи, когда поведение намеренно отделяется от класса, содержащего данные. Чаще всего это делают для того, чтобы иметь возможность динамически менять это поведение (паттерны Стратегия, Посетитель и т.д.).

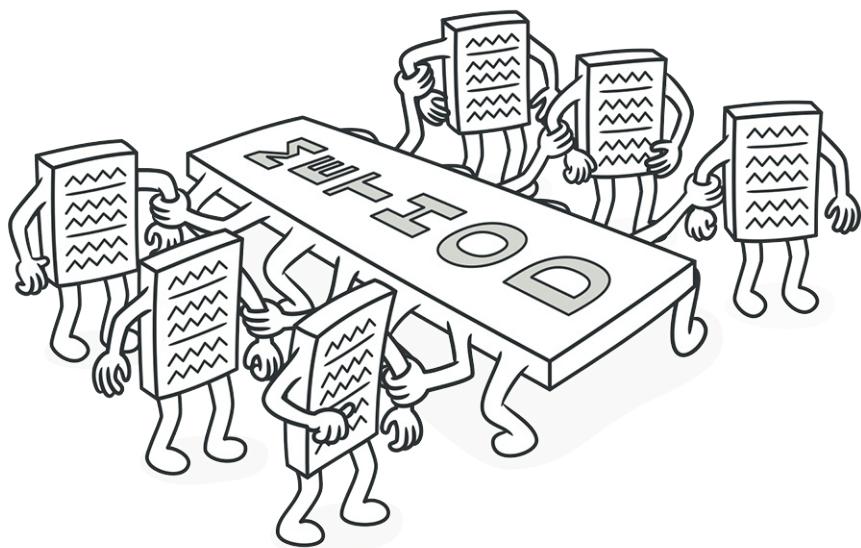


# Неуместная близость

Также известен как *Inappropriate Intimacy*

## Симптомы и признаки

Один класс использует служебные поля и методы другого класса.



## Причины появления

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.

## Лечение

- Самый простой выход – при помощи перемещения метода и перемещения 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.



- Другим решением является извлечение зависимых частей в отдельный класс и скрытие делегирования к этому классу.
- Если между классами существует обоюдная зависимость, стоит прибегнуть к замене двунаправленной связи на одностороннюю.
- Если близость возникает между подклассом и родительским классом, рассмотрите возможность замены делегирования наследованием.

## Выигрыш

- Улучшает организацию кода.
- Упрощает техническую поддержку и повторное использование кода.



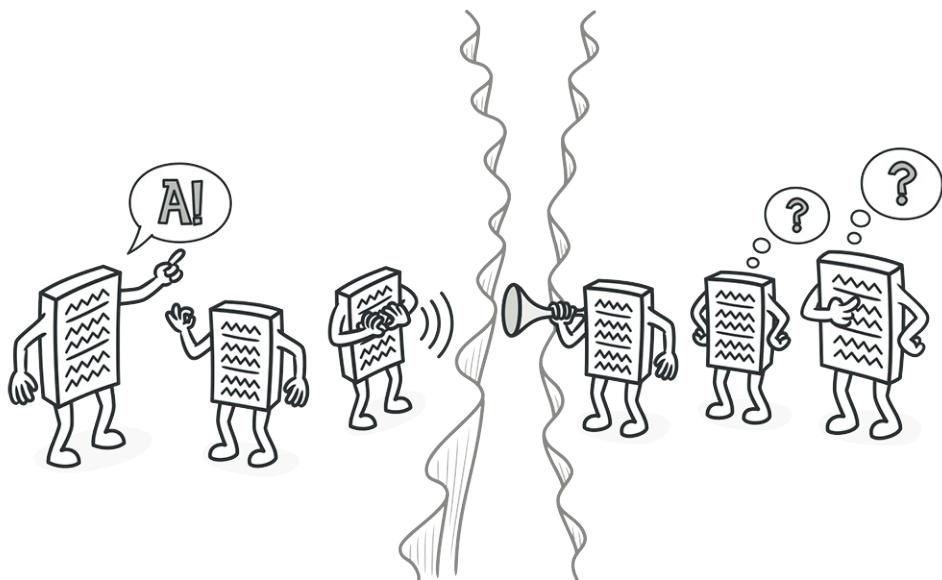


# Цепочка вызовов

Также известен как *Message Chains*

## Симптомы и признаки

Вы видите в коде цепочки вызовов вроде такой `$a->b () ->c () ->d ()`



## Причины появления

Цепочка вызовов появляется тогда, когда клиент запрашивает у одного объекта другой, в свою очередь этот объект запрашивает ещё один и т. д. Такие последовательности вызовов означают, что клиент связан с навигацией по структуре классов. Любые изменения промежуточных связей означают необходимость модификации клиента.

## Лечение

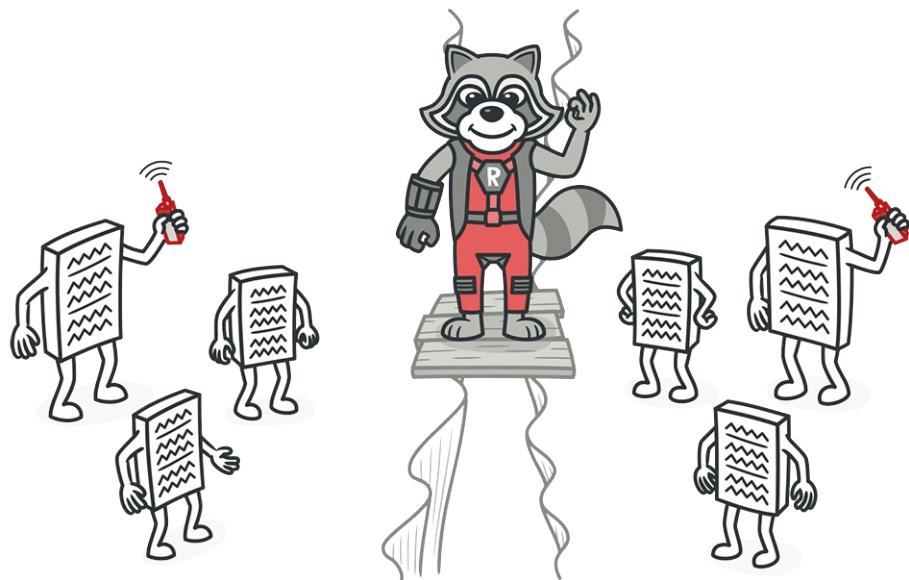
- Для удаления цепочки вызовов применяется приём сокрытие делегирования.



- Иногда лучше рассмотреть, для чего используется конечный объект. Может быть, имеет смысл использовать **извлечение метода**, чтобы извлечь эту функциональность, и передвинуть её в самое начало цепи с помощью **перемещения метода**.

## Выигрыш

- Может уменьшить связность между классами цепочки.
- Может уменьшить размер кода.



**Не стоит трогать, если...**

- Если вы перестааетесь в процессе сокрытия делегирования, в коде будет довольно сложно понять, где именно осуществляется конкретная работа. Другими словами, появится запах Посредник.

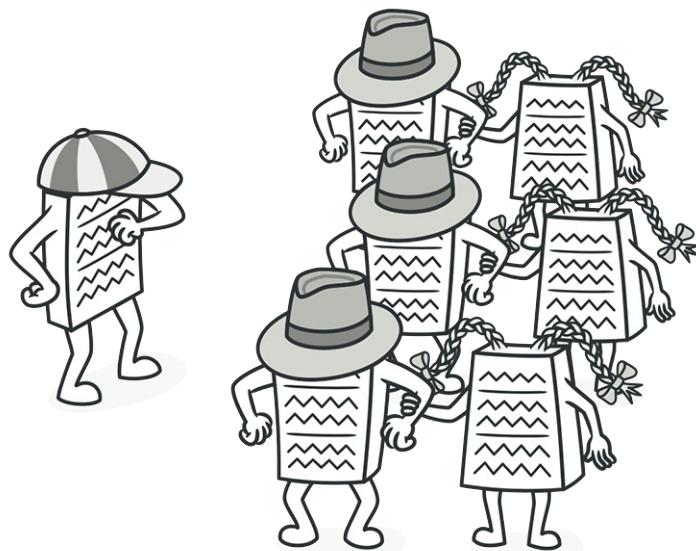


# **Посредник**

Также известен как *Middle Man*

## Симптомы и признаки

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



## Причины появления

Данный запах может быть результатом фанатичной борьбы с цепочками вызовов.

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

## Лечение

- Если большую часть методов класс делегирует другому классу, нужно воспользоваться удалением посредника.

## Выигрыш

- Уменьшение размера кода.



## Не стоит трогать, если...

Не удаляйте посредников, которые были созданы осознанно:

- Посредник мог быть введён для избавления от нежелательной зависимости между классами.
- Некоторые паттерны проектирования намеренно создают посредников (например, Заместитель или Декоратор).



# Остальные запахи

Ниже приведены особые запахи, которые не попали ни в одну обширную категорию.

## § Неполнота библиотечного класса

**Библиотеки** через некоторое время перестают удовлетворять требованиям пользователей. Естественное решение – внести изменения в библиотеку – очень часто оказывается недоступным, т.к. библиотека закрыта для записи.

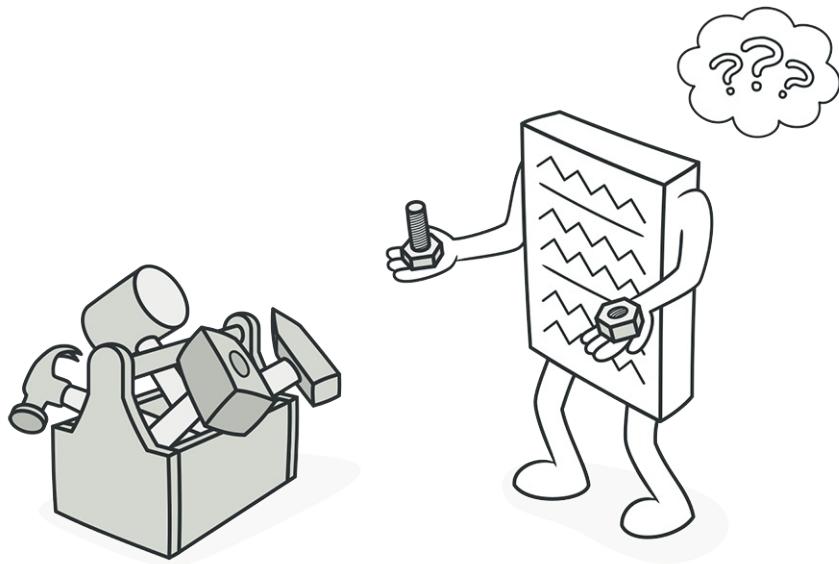


# Неполнота библиотечного класса

Также известен как *Incomplete Library Class*

## Симптомы и признаки

**Библиотеки** через некоторое время перестают удовлетворять требованиям пользователей. Естественное решение – внести изменения в библиотеку – очень часто оказывается недоступным, т.к. библиотека закрыта для записи.



## Причины появления

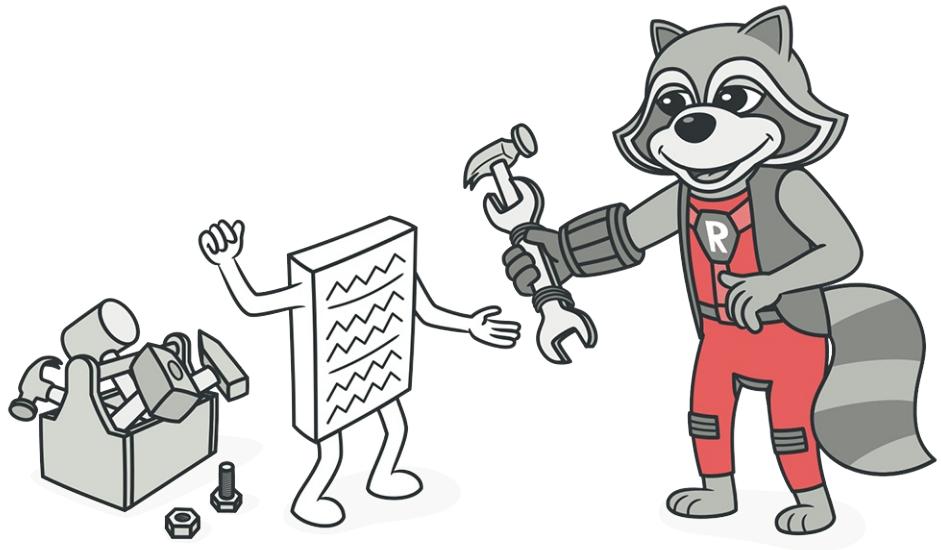
Автор библиотеки не предусмотрел возможности, которые вам нужны, либо отказался их внедрять.

## Лечение

- Если надо добавить пару методов в библиотечный класс, используется **введение внешнего метода**.
- Если надо серьёзно поменять поведение класса, используется **введение локального расширения**.

## Выигрыш

- Уменьшает дублирование кода (вместо создания своей библиотеки с нуля, вы используете готовую библиотеку).



## Не стоит трогать, если...

- Расширение библиотеки может стать причиной появления дополнительного объема работы. Это происходит в том случае, когда изменения в библиотеке затрагивают изменения в коде.



# Техники рефакторинга

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.

Техники рефакторинга описывают конкретные методы борьбы с грязным кодом. Большинство рефакторингов имеет как достоинства, так и недостатки. Поэтому любой рефакторинг должен быть мотивирован и обдуман.

В предыдущих главах вы увидели как те или иные рефакторинги можно применить для лечения проблем с кодом. Самое время рассмотреть конкретные приёмы рефакторинга в деталях!



# Составление методов

Значительная часть рефакторинга посвящается правильному составлению методов. В большинстве случаев, корнем всех зол являются слишком длинные методы. Хитросплетения кода внутри такого метода, прячут логику выполнения и делают метод крайне сложным для понимания, а значит и изменения.

Рефакторинги этой группы призваны уменьшить сложностью внутри метода, убрать дублирование кода и облегчить последующую работу с ним.

## § Извлечение метода

**Проблема:** У вас есть фрагмент кода, который можно сгруппировать.

**Решение:** Выделите участок кода в новый метод (или функцию) и вызовите этот метод вместо старого кода.

## § Встраивание метода

**Проблема:** Стоит использовать в том случае, когда тело метода очевиднее самого метода.

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

## § Извлечение переменной

**Проблема:** У вас есть сложное для понимания выражение.

**Решение:** Поместите результат выражения или его части в отдельные переменные, поясняющие суть выражения.

## § Встраивание переменной

**Проблема:** У вас есть временная переменная, которой присваивается результат простого выражения (и больше ничего).

**Решение:** Замените обращения к переменной этим выражением.

## § Замена переменной вызовом метода

**Проблема:** Вы помещаете результат какого-то выражения в локальную переменную, чтобы использовать её далее в коде.

**Решение:** Выделите все выражение в отдельный метод и возвращайте результат из него. Замените использование вашей переменной вызовом метода. Новый метод может быть использован и в других методах.

## § Расщепление переменной

**Проблема:** У вас есть локальная переменная, которая используется для хранения разных промежуточных значений внутри метода (за исключением переменных циклов).

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

## § Удаление присваиваний параметрам

**Проблема:** Параметру метода присваивается какое-то значение.

**Решение:** Вместо параметра воспользуйтесь новой локальной переменной.

## § Замена метода объектом методов

**Проблема:** У вас есть длинный метод, в котором локальные переменные так сильно переплетены, что это делает невозможным применение **извлечения метода**.

**Решение:** Преобразуйте метод в отдельный класс так, чтобы локальные переменные стали полями этого класса. После этого можно без труда разделить метод на части.

## § Замена алгоритма

**Проблема:** Вы хотите заменить существующий алгоритм другим?

**Решение:** Замените тело метода, реализующего старый алгоритм, новым алгоритмом.



# ✂ Извлечение метода

Также известен как *Extract Method*

## Проблема

У вас есть фрагмент кода, который можно сгруппировать.

```
void printOwing() {  
    printBanner();  
  
    //print details  
    System.out.println("name: " + name);  
    System.out.println("amount: " + getOutstanding());  
}
```

## Решение

Выделите участок кода в новый метод (или функцию) и вызовите этот метод вместо старого кода.

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

## Причины рефакторинга

Чем больше строк кода в методе, тем сложнее разобраться в том, что он делает. Это основная проблема, которую решает этот рефакторинг.

Извлечение метода не только убивает множество запашков в коде, но и является одним из этапов множества других рефакторингов.

# Достоинства

- Улучшает читабельность кода. Постарайтесь дать новому методу название, которое бы отражало суть того, что он делает. Например, `createOrder()`, `renderCustomerInfo()` и т.д.
- Убирает дублирование кода. Иногда код, вынесенный в метод, можно найти и в других местах программы. В таком случае, имеет смысл заменить найденные участки кода вызовом вашего нового метода.
- Изолирует независимые части кода, уменьшая вероятность ошибок (например, по вине переопределения не той переменной).

## Порядок рефакторинга

1. Создайте новый метод и назовите его так, чтобы название отражало суть того, что будет делать этот метод.
2. Скопируйте беспокоящий вас фрагмент кода в новый метод. Удалите этот фрагмент из старого места и замените вызовом вашего нового метода.

Найдите все переменные, которые использовались в этом фрагменте кода. Если они были объявлены внутри этого фрагмента и не используются вне его, просто оставьте их без изменений – они станут локальными переменными нового метода.

3. Если переменные объявлены перед интересующим вас участком кода, значит, их следует передать в параметры вашего нового метода, чтобы использовать значения, которые в них находились ранее. Иногда от таких переменных проще избавиться с помощью замены переменных вызовом метода.
4. Если вы видите, что локальная переменная как-то изменяется в вашем участке кода, это может означать, что её изменённое значение понадобится дальше в основном методе. Проверьте это. Если подозрение подтвердилось, значение этой переменной следует возвратить в основной метод, чтобы ничего не сломать.

## Анти-рефакторинг

### § Встраивание метода



## Родственные рефакторинги

### § Перемещение метода



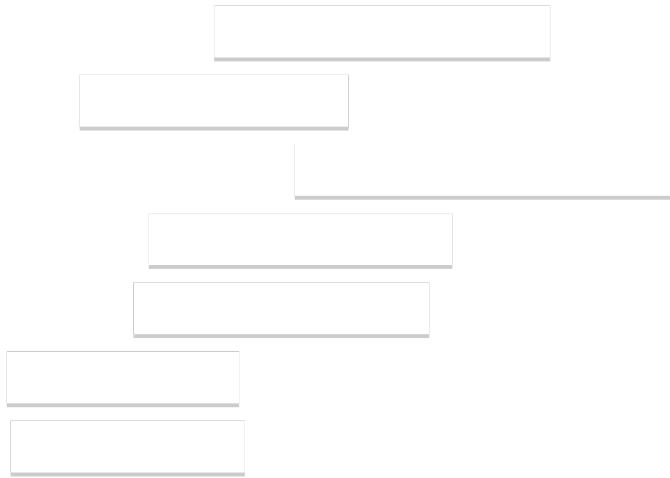
# Помогает рефакторингу

- § Замена параметров объектом
- § Создание шаблонного метода
- § Замена параметров объектом
- § Параметризация метода



# Борется с запахом

- § Дублирование кода
- § Длинный метод
- § Завистливые функции
- § Операторы switch
- § Цепочка вызовов
- § Комментарии
- § Класс данных





# ✂ Встраивание метода

Также известен как *Inline Method*

## Проблема

Стоит использовать в том случае, когда тело метода очевиднее самого метода.

```
class PizzaDelivery {  
    //...  
  
    int getRating() {  
        return moreThanFiveLateDeliveries() ? 2 : 1;  
    }  
  
    boolean moreThanFiveLateDeliveries() {  
        return numberOfLateDeliveries > 5;  
    }  
}
```

## Решение

Замените вызовы метода его содержимым и удалите сам метод.

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

## Причины рефакторинга

Основная причина – тело метода состоит из простого делегирования к другому методу. Само по себе такое делегирование – не проблема. Но если таких методов довольно много, становится очень легко в них запутаться.

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

## Достоинства

- Минимизируя количество бесполезных методов, мы уменьшаем общую сложность кода.

## Порядок рефакторинга

1. Убедитесь, что метод не переопределяется в подклассах. Если он переопределяется, воздержитесь от рефакторинга.
2. Найдите все вызовы метода. Замените эти вызовы содержимым метода.
3. Удалите метод.

## Анти-рефакторинг

### § Извлечение метода



## Борется с запахом

### § Теоретическая общность





# Извлечение переменной

Также известен как *Extract Variable*

## Проблема

У вас есть сложное для понимания выражение.

```
void renderBanner() {  
    if ((platform.toUpperCase().indexOf("MAC") > -1) &&  
        (browser.toUpperCase().indexOf("IE") > -1) &&  
        wasInitialized() && resize > 0)  
    {  
        // do something  
    }  
}
```

## Решение

Поместите результат выражения или его части в отдельные переменные, поясняющие суть выражения.

```
void renderBanner() {  
  
    final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;  
    final boolean isIE = browser.toUpperCase().indexOf("IE") > -1;  
    final boolean wasResized = resize > 0;  
  
    if (isMacOs && isIE && wasInitialized() && wasResized) {  
        // do something  
    }  
}
```

## Причины рефакторинга

Главная мотивация этого рефакторинга – сделать более понятным сложное выражение, разбив его на промежуточные части.

Это может быть:

- Условие оператора `if()` или части оператора `?:` в С-подобных языках.
- Длинное арифметическое выражение без промежуточных результатов.
- Длинное склеивание строк.

Выделение переменной может стать первым шагом к последующему извлечению метода, если вы увидите, что выделенное выражение используется и в других местах кода.

## Достоинства

- Улучшает читабельность кода. Постарайтесь дать выделенным переменным хорошие названия, которые будут отражать точно суть выражения. Так вы сделаете код читабельным и сумеете избавиться от лишних комментариев. Например, `customerTaxValue`, `cityUnemploymentRate`, `clientSalutationString` и т.д.

## Недостатки

- Появляются дополнительные переменные. Но этот минус компенсируется простотой чтения кода.

## Порядок рефакторинга

1. Вставьте новую строку перед интересующим вас выражением и объявите там новую переменную. Присвойте этой переменной часть сложного выражения.
2. Замените часть вынесенного выражения новой переменной.
3. Повторите это для всех сложных частей выражения.

## Анти-рефакторинг

### § Встраивание переменной

## Родственные рефакторинги

### § Извлечение метода

## Борется с запахом

### § Комментарии



# ☒ Встраивание переменной

Также известен как *Inline Temp*

## Проблема

У вас есть временная переменная, которой присваивается результат простого выражения (и больше ничего).

```
double hasDiscount(Order order) {  
    double basePrice = order.basePrice();  
    return (basePrice > 1000);  
}
```

## Решение

Замените обращения к переменной этим выражением.

```
double hasDiscount(Order order) {  
    return (order.basePrice() > 1000);  
}
```

## Причины рефакторинга

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

## 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 с виду бесполезные 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.

# Порядок рефакторинга

1. Найдите все места, где используется переменная, и замените их выражением, которое ей присваивалось.
2. Удалите объявление переменной и строку присваивания ей значения.

## Помогает рефакторингу

§ Замена переменной вызовом метода

§ Извлечение метода



# ☒ Замена переменной вызовом метода

Также известен как *Replace Temp with Query*

## Проблема

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

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

## Решение

Выделите все выражение в отдельный метод и возвращайте результат из него. Замените использование вашей переменной вызовом метода. Новый метод может быть использован и в других методах.

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

# Причины рефакторинга

Применение данного рефакторинга может быть подготовительным этапом для применения выделения метода для какой-то части очень длинного метода.

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

## Достоинства

- Улучшает читабельность кода. Намного проще понять, что делает метод `getTax()` чем строка `orderPrice() * -2`.
- Помогает убрать дублирование кода, если заменяемая строка используется более чем в одном методе.

## Полезные факты

### Вопрос производительности

При использовании этого рефакторинга может возникнуть вопрос, не скажется ли результат рефакторинга на лучшим образом на производительности программы. Честный ответ – да, результирующий код может получить дополнительную нагрузку за счёт вызова нового метода. Однако в наше время быстрых процессоров и хороших компиляторов такая нагрузка вряд ли будет заметна. Зато взамен мы получаем лучшую читабельность кода и возможность использовать новый метод в других местах программы.

Тем не менее, если ваша временная переменная служит для кеширования результата действительно трудоёмкого выражения, имеет смысл остановить этот рефакторинг после выделения выражения в новый метод.

## Порядок рефакторинга

1. Убедитесь, что переменной в пределах метода присваивается значение только один раз. Если это не так, используйте разделение переменной для того, чтобы гарантировать, что переменная будет использована только для хранения результата вашего выражения.
2. Используйте извлечение метода для того, чтобы переместить интересующее вас выражение в новый метод. 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.
3. Замените использование переменной вызовом вашего нового метода.

# Родственные рефакторинги

§ Извлечение метода

## Борется с запахом

§ Длинный метод

§ Дублирование кода



# Расщепление переменной

Также известен как *Split Temporary Variable*

## Проблема

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

```
double temp = 2 * (height + width);  
System.out.println(temp);  
temp = height * width;  
System.out.println(temp);
```

## Решение

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

```
final double perimeter = 2 * (height + width);  
System.out.println(perimeter);  
final double area = height * width;  
System.out.println(area);
```

## Причины рефакторинга

Если вы «экономите» переменные внутри функции, повторно используете их для различных несвязанных целей, у вас обязательно начнутся проблемы в тот момент, когда потребуется внести какие-то изменения в код, содержащий эти переменные. Вам придётся перепроверять все случаи использования переменной, чтобы удостовериться в отсутствии ошибки в коде.

## Достоинства

- Каждый элемент программы должен отвечать только за одну вещь. Это сильно упрощает поддержку кода в будущем, т.к. вы можете спокойно заменить этот элемент, не опасаясь побочных эффектов.
- Улучшается читабельность кода. Если переменная создавалась очень давно, да еще и в спешке, она могла получить элементарное название, которое не объясняет сути хранимого значения, например, `k`, `a2`, `value` и т.д. У вас есть

шанс исправить ситуацию, назначив новым переменным хорошие названия, отражающие суть хранимых значений. Например, `customerTaxValue`, `cityUnemploymentRate`, `clientSalutationString` и т.д.

- Этот рефакторинг помогает в дальнейшем выделить повторяющиеся участки кода в отдельные методы.

## Порядок рефакторинга

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.

## Анти-рефакторинг

- § Встраивание переменной

## Родственные рефакторинги

- § Извлечение переменной

- § Удаление присваиваний параметрам

## Помогает рефакторингу

- § Извлечение метода



# ✖ Удаление присваиваний параметрам

Также известен как *Remove Assignments to Parameters*

## Проблема

Параметру метода присваивается какое-то значение.

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

## Решение

Вместо параметра воспользуйтесь новой локальной переменной.

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

## Причины рефакторинга

Причины проведения этого рефакторинга такие же, как и при расщеплении переменной, но в данном случае речь идёт о параметре, а не о локальной переменной.

Во-первых, если параметр передаётся по ссылке, то после изменения его значения внутри метода, оно передается аргументу, который подавался на вызов этого метода. Очень часто это происходит случайно и приводит к печальным последствиям. Даже если в вашем языке программирования параметры обычно передаются по значению, а не по ссылке, сам код может вызвать замешательство у тех, кто привык считать иначе.

Во-вторых, множественные присваивания разных значений параметру приводят к тому, что вам становится сложно понять, какие именно данные должны находиться в параметре в определенный момент времени. Проблема усугубляется, если ваш параметр и то, что он должен хранить, описаны в документации, но фактически, его значение может не совпадать с ожидаемым внутри метода.

## Достоинства

- Каждый элемент программы должен отвечать только за одну вещь. Это сильно упрощает поддержку кода в будущем, т.к. вы можете спокойно заменить этот элемент, не опасаясь побочных эффектов.
- Этот рефакторинг помогает в дальнейшем **выделить повторяющиеся участки кода в отдельные методы.**

## Порядок рефакторинга

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

## Родственные рефакторинги

### § **Расщепление переменной**

## Помогает рефакторингу

### § **Извлечение метода**



# ☒ Замена метода объектом методов

Также известен как *Replace Method with Method Object*

## Проблема

У вас есть длинный метод, в котором локальные переменные так сильно переплетены, что это делает невозможным применение **извлечения метода**.

```
class Order {  
    //...  
    public double price() {  
        double primaryBasePrice;  
        double secondaryBasePrice;  
        double tertiaryBasePrice;  
        // long computation.  
        //...  
    }  
}
```

## Решение

Преобразуйте метод в отдельный класс так, чтобы локальные переменные стали полями этого класса. После этого можно без труда разделить метод на части.

```
class Order {  
    //...  
  
    public double price() {  
        return new PriceCalculator(this).compute();  
    }  
}  
  
class PriceCalculator {  
  
    private double primaryBasePrice;  
    private double secondaryBasePrice;  
    private double tertiaryBasePrice;  
  
    public PriceCalculator(Order order) {  
        // copy relevant information from order object.  
        //...  
    }  
  
    public double compute() {  
        // long computation.  
        //...  
    }  
}
```

## Причины рефакторинга

Метод слишком длинный, и вы не можете его разделить из-за хитросплетения локальных переменных, которые сложно изолировать друг от друга.

Первым шагом к решению проблемы будет выделение всего этого метода в отдельный класс и превращение его локальных переменных в поля.

Во-первых, это позволит вам изолировать проблему в пределах этого класса, а во-вторых, расчистит дорогу для разделения большого метода на методы поменьше, которые, к тому же, не подходили бы к смыслу оригинального класса.

## Достоинства

- Изоляция длинного метода в собственном классе позволяет остановить бесконтрольный рост метода. Кроме того, даёт возможность разделить его на подметоды в рамках своего класса, не засоряя служебными методами оригинальный класс.

## Недостатки

- Создаётся ещё один класс, повышая общую сложность программы.

## Порядок рефакторинга

1. Создайте новый класс. Дайте ему название, основываясь на предназначении метода, который рефакторите.
2. В новом классе создайте приватное поле для хранения ссылки на экземпляр класса, в котором раньше находился метод. Эту ссылку потом можно будет использовать, чтобы брать из оригинального объекта нужные данные, если потребуется.
3. Создайте отдельное приватное поле для каждой локальной переменной метода.
4. Создайте конструктор, который принимает в параметрах значения всех локальных переменных метода, а также инициализирует соответствующие приватные поля.
5. Объявите основной метод и скопируйте в него код оригинального метода, заменив локальные переменные приватным полями.
6. Замените тело оригинального метода в исходном классе созданием объекта-метода и вызовом его основного метода.

## Родственные рефакторинги

### § Замена простого поля объектом

| Делает то же, но с полем.

## Борется с запахом

### § Длинный метод



# ⌘ Замена алгоритма

Также известен как *Substitute Algorithm*

## Проблема

Вы хотите заменить существующий алгоритм другим?

```
String foundPerson(String[] people) {  
    for (int i = 0; i < people.length; i++) {  
        if (people[i].equals("Don")) {  
            return "Don";  
        }  
        if (people[i].equals("John")) {  
            return "John";  
        }  
        if (people[i].equals("Kent")) {  
            return "Kent";  
        }  
    }  
    return "";  
}
```

## Решение

Замените тело метода, реализующего старый алгоритм, новым алгоритмом.

```
String foundPerson(String[] people) {  
    List candidates =  
        Arrays.asList(new String[] {"Don", "John", "Kent"});  
    for (int i=0; i < people.length; i++) {  
        if (candidates.contains(people[i])) {  
            return people[i];  
        }  
    }  
    return "";  
}
```

## Причины рефакторинга

1. Поэтапный рефакторинг – не единственный способ улучшить программу. Иногда вы сталкиваетесь с таким нагромождением проблем в методе, что его гораздо проще переписать заново. С другой стороны, вы могли найти алгоритм, который куда проще и эффективнее текущего. В этом случае надо просто заменить старый алгоритм новым.
2. С течением времени ваш алгоритм может оказаться включен в набор известной библиотеки или фреймворка, и вы можете пожелать избавиться от собственной реализации, чтобы облегчить себе поддержку программы.
3. Требования к работе программы могут измениться настолько сильно, что старый алгоритм невозможно просто «допилить» до соответствия им.

## Порядок рефакторинга

1. Убедитесь, что вы по максимуму упростили текущий алгоритм. Перенесите несущественный код в другие методы с помощью извлечения метода. Чем меньше «движущихся частей» останется в исходном алгоритме, тем проще будет его заменить.
2. Создайте ваш новый алгоритм в новом методе. Замените старый алгоритм новым и запустите тесты программы.
3. Если результаты не сходятся, верните старую реализацию и сравните результаты. Выясните, почему результаты не совпадают. Нередко, причиной могут быть ошибки в старом алгоритме, но с большей вероятностью не работает что-то в новом.
4. Когда все тесты начнут проходить, окончательно удалите старый алгоритм.

# Борется с запахом

§ Дублирование кода

§ Длинный метод



# Перемещение функций между объектами

Если вы разместили функциональность по классам не самым удачным образом – это еще не повод отчаиваться.

Рефакторинги этой группы показывают как безопасно перемещать функциональность из одних классов в другие, создавать новые классы, а также скрывать детали реализации из публичного доступа.

## § Перемещение метода

**Проблема:** Метод используются в другом классе больше, чем в собственном.

**Решение:** 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.

## § Перемещение поля

**Проблема:** Поле используется в другом классе больше, чем в собственном.

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

## § Извлечение класса

**Проблема:** Один класс работает за двоих.

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

## § Встраивание класса

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

**Решение:** Переместите все фичи из описанного класса в другой.

## § Скрытие делегирования

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

**Решение:** Создайте новый метод в классе А, который бы делегировал вызов объекту В. Таким образом, клиент перестанет знать о классе В и зависеть от него.

## § Удаление посредника

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

**Решение:** Удалите эти методы и заставьте клиента вызывать конечные методы напрямую.

## § Введение внешнего метода

**Проблема:** Служебный класс не содержит метода, который вам нужен, при этом у вас нет возможности добавить метод в этот класс.

**Решение:** Добавьте метод в клиентский класс и передавайте в него объект служебного класса в качестве аргумента.

## § Введение локального расширения

**Проблема:** В служебном классе отсутствуют некоторые методы, которые вам нужны. При этом добавить их в этот класс вы не можете.

**Решение:** Создайте новый класс, который бы содержал эти методы, и сделайте его наследником служебного класса, либо его обёрткой.

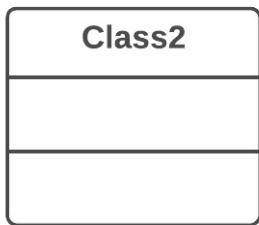
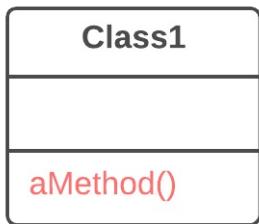


# ✂ Перемещение метода

Также известен как *Move Method*

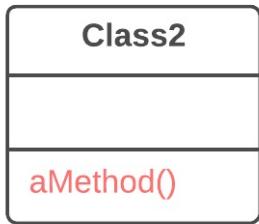
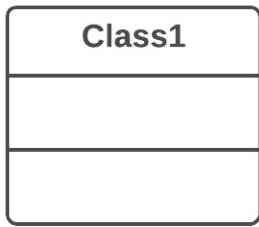
## Проблема

Метод используются в другом классе больше, чем в собственном.



## Решение

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.



## Причины рефакторинга

1. Вы хотите переместить метод в класс, который содержит данные, с которыми, в основном, работает этот метод. Этим вы **увеличиваете связность внутри классов**.
2. Вы хотите переместить метод, чтобы убрать или уменьшить зависимость класса, вызывающего этот метод, от класса, в котором он находился. Это может быть полезно, если вызывающий класс уже имеет зависимость от класса, куда вы планируете перенести метод. Таким образом, вы **уменьшаете связанность между классами**.

## Порядок рефакторинга

1. Проверьте все фичи, используемые старым методом в его же классе. Возможно, их тоже следует переместить. Руководствуйтесь таким правилом – если фича используется только интересующим вас методом, её точно следует переносить. Если фича используется и другими методами, возможно, следует перенести и эти методы. Иногда гораздо проще перенести пачку методов, чем настраивать взаимодействие между ними в разных классах.

Проверьте, не определён ли метод в суперклассах и подклассах приемника. Если это так, вам придётся либо отказаться от идеи переноса, либо реализовать в классе-получателе подобие полиморфизма, чтобы обеспечить различную функциональность метода, которая была разнесена по классам-донорам.

2. Объявите новый метод в классе-приёмнике. Возможно, следует придумать для метода новое имя, которое в новом классе будет подходить ему больше.

5. Определите, как вы будете обращаться к классу-получателю. Вполне возможно, у вас уже есть поле или метод, которые возвращают подходящий объект, но если нет, нужно будет написать новый метод или поле, в котором бы хранился объект класса-получателя.

Теперь у вас есть способ обратиться к объекту-получателю и новый метод в его классе. С этим всем вы уже можете превратить старый метод в обращение к новому методу.

4. Оцените, есть ли возможность удалить старый метод вообще? При этом нужно будет во всех местах, где используется этот метод, поставить обращение к новому.

## Родственные рефакторинги

§ Извлечение метода

§ Перемещение поля

## Помогает рефакторингу

§ Извлечение класса

§ Встраивание класса

§ Замена параметров объектом

## Борется с запахом

§ Стрельба дробью

§ Завистливые функции

§ Операторы switch

§ Параллельные иерархии наследования

§ Цепочка вызовов

§ Неуместная близость

§ Класс данных

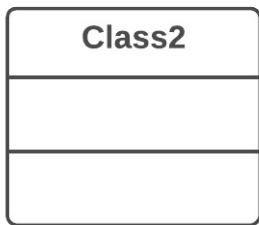


# ✂ Перемещение поля

Также известен как *Move Field*

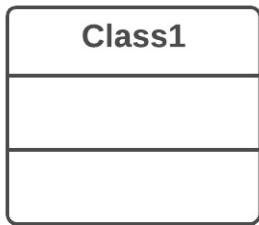
## Проблема

Поле используется в другом классе больше, чем в собственном.



## Решение

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



## Причины рефакторинга

Зачастую поля переносятся как часть извлечение одного класса из другого. Решить, в каком из классов должно остаться поле, бывает непросто. Тем не менее, у нас есть неплохой рецепт — **поле должно быть там, где находятся методы, которые его используют** (либо там, где этих методов больше).

Это правило поможет вам и в других случаях, когда поле попросту находится не там, где нужно.

## Порядок рефакторинга

1. Если поле публичное, вам будет намного проще совершить рефакторинг, если вы сделаете его приватным и предоставите публичные методы доступа (для этого можно использовать рефакторинг инкапсуляция поля).
2. Создайте такое же поле с методами доступа в классе-приёмнике.
3. Определите, как вы будете обращаться к классу-получателю. Вполне возможно, у вас уже есть поле или метод, которые возвращают подходящий объект. Если нет — нужно будет написать новый метод или поле, в котором бы хранился объект класса-получателя.
4. Замените все обращения к старому полю на соответствующие вызовы методов в классе-получателе. Если поле не приватное, проделайте это и в суперклассе, и в подклассах.
5. Удалите поле в исходном классе.

## Родственные рефакторинги

§ Перемещение поля

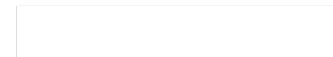


## Помогает рефакторингу

§ Извлечение класса



§ Встраивание класса

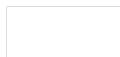


## Борется с запахом

§ Стрельба дробью



§ Параллельные иерархии наследования



§ Неуместная близость



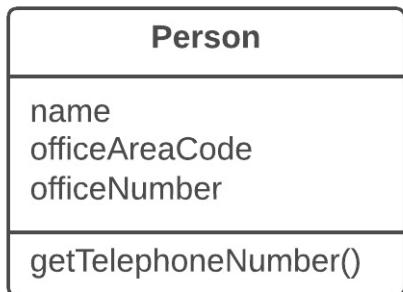


# Извлечение класса

Также известен как *Extract Class*

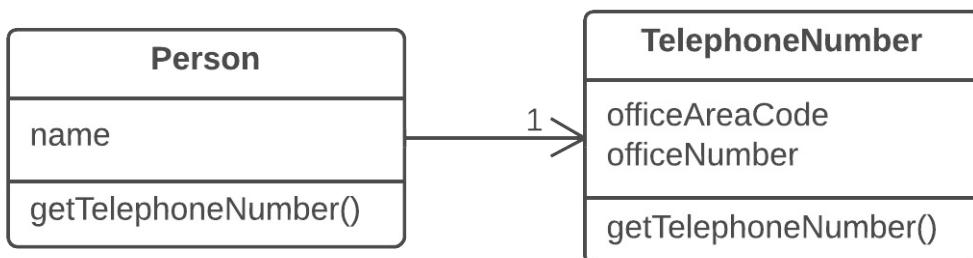
## Проблема

Один класс работает за двоих.



## Решение

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



## Причины рефакторинга

Классы всегда изначально выглядят чёткими и понятными. Они выполняют свою работу и не лезут в обязанности других классов. Однако, с течением жизни программы добавляется один метод – тут, одно поле – там. В результате некоторые классы получают массу дополнительных обязанностей.

## Достоинства

- Этот рефакторинг призван помочь в соблюдении принципа *единственной обязанности класса*. Это делает код ваших классов очевиднее и понятнее.
- Классы с единственной обязанностью более надёжны и устойчивы к изменениям. Например, у вас есть класс, отвечающий за десять разных вещей. И когда вам придётся вносить в него изменения, вы рискуете при корректировках одной вещи сломать другие.

## Недостатки

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

## Порядок рефакторинга

Перед началом рефакторинга обязательно определите, как именно следует разделить обязанности класса.

1. Создайте новый класс, который будет содержать выделенную функциональность.
2. Создайте связь между старым и новым классом. Лучше всего, если эта связь будет односторонней; при этом второй класс можно будет без проблем использовать повторно. С другой стороны, если вы считаете, что это необходимо, всегда можно создать двустороннюю связь.
3. Используйте перемещение поля и перемещение метода для каждого поля и метода, которые вы решили переместить в новый класс. Для методов имеет смысл начинать с приватных, таким образом вы снижаете вероятность допустить массу ошибок. Страйтесь двигаться понемногу и тестировать результат после каждого перемещения, это избавит вас от необходимости исправлять большое число ошибок в самом конце.

После того как с перемещением покончено, пересмотрите ещё раз на получившиеся классы. Возможно, старый класс теперь имеет смысл назвать по-другому ввиду его изменившихся обязанностей. Проверьте ещё раз, можно ли избавиться от двусторонней связи между классами, если она возникла.

4. Ещё одним нюансом является доступность нового класса извне. Вы можете полностью спрятать его от клиента, сделав приватным, управляя при этом его полями из старого класса. Либо сделать его публичным, предоставив клиенту возможность напрямую менять значения. Решение зависит от того, насколько безопасны для поведения старого класса будут неожиданные прямые изменения значений в новом классе.

# Анти-рефакторинг

§ Встраивание класса

## Родственные рефакторинги

§ Извлечение подкласса

§ Замена простого поля объектом

## Борется с запахом

§ Дублирование кода

§ Большой класс

§ Расходящиеся модификации

§ Группы данных

§ Одержанность элементарными типами

§ Временное поле

§ Неуместная близость

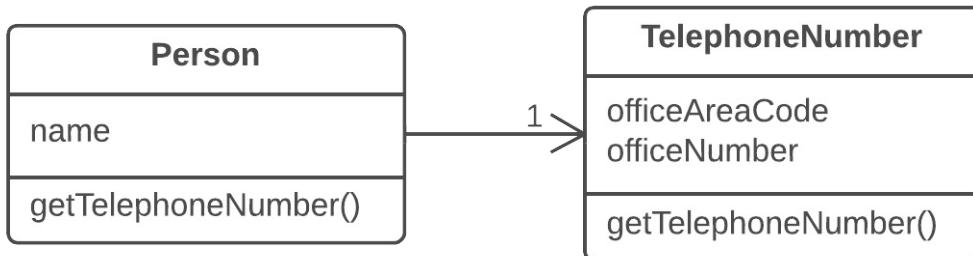


# ✂ Встраивание класса

Также известен как *Inline Class*

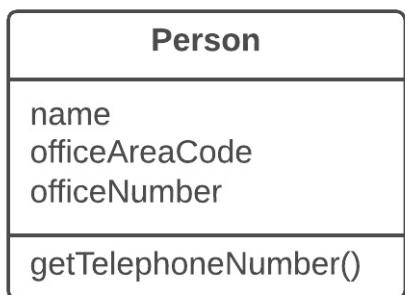
## Проблема

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



## Решение

Переместите все фичи из описанного класса в другой.



## 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.

# Порядок рефакторинга

1. Создайте в классе-приёмнике публичные поля и методы, которые есть в классе-доноре. Методы должны обращаться к аналогичным методам класса-донора.
2. Замените все обращения к классу-донору обращениями к полям и методам класса-приёмника.
3. Самое время протестировать программу и убедиться, что во время работы не было допущено никаких ошибок. Если тесты показали, что все работает так, как должно, начинаем использовать **перемещение метода** и **перемещение поля** для того, чтобы полностью переместить все функциональности в класс-приёмник из исходного класса. Продолжаем делать это, пока исходный класс не окажется совсем пустым.
4. Удалите исходный класс.

# Анти-рефакторинг

## § Извлечение класса

## Борется с запахом

### § Стрельба дробью

### § Ленивый класс

### § Теоретическая общность

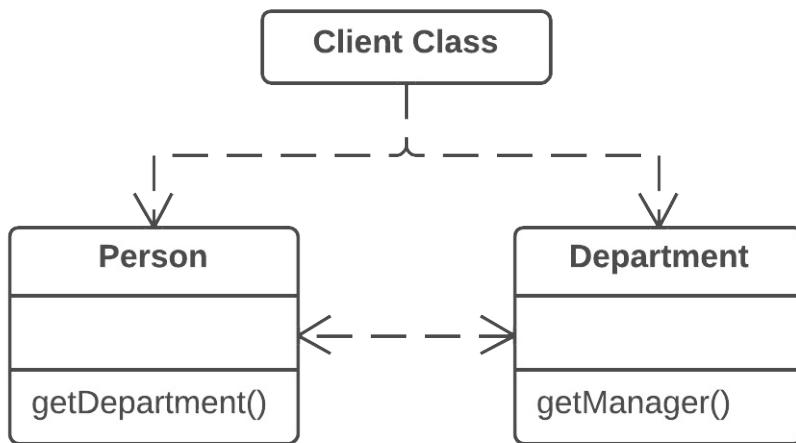


# 🚫 Сокрытие делегирования

Также известен как *Hide Delegate*

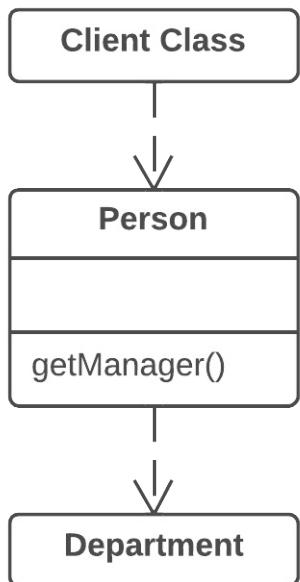
## Проблема

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



## Решение

Создайте новый метод в классе А, который бы делегировал вызов объекту В. Таким образом, клиент перестанет знать о классе В и зависеть от него.



# Причины рефакторинга

Для начала следует определиться с названиями:

- *делегат* – это конечный объект, который содержит функциональность, нужную клиенту;
- *сервер* – это объект, к которому клиент имеет непосредственный доступ.

Цепочка вызовов появляется тогда, когда клиент запрашивает у одного объекта другой, потом второй объект запрашивает еще один и т.д. Такие последовательности вызовов означают, что 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* рискует превратиться в лишнее промежуточное звено и привести к запашку посредник.

## Порядок рефакторинга

1. Для каждого метода *класса-делегата*, вызываемого клиентом, нужно создать метод в *классе-сервере*, который бы делегировал вызов *классу-делегату*.
2. Измените код клиента так, чтобы он вызывал методы *класса-сервера*.
3. Если после всех изменений клиент больше не нуждается в *классе-делегате*, можно убрать метод доступа к *классу-делегату* из *класса-сервера* (тот метод, который использовался изначально для получения *класса-делегата*).

## Анти-рефакторинг

### § Удаление посредника

## Борется с запахом

§ Цепочка вызовов

§ Неуместная близость

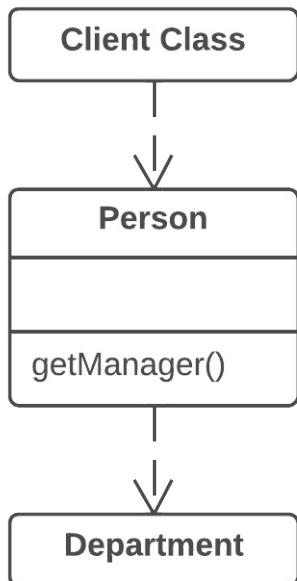


# ✂ Удаление посредника

Также известен как *Remove Middle Man*

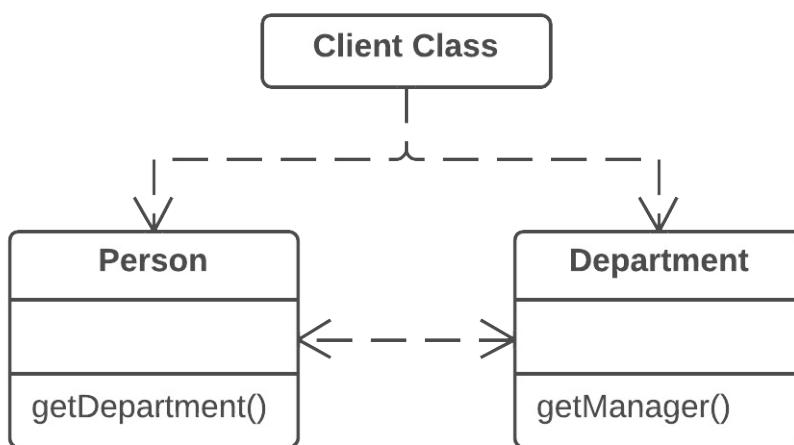
## Проблема

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



## Решение

Удалите эти методы и заставьте клиента вызывать конечные методы напрямую.



# Причины рефакторинга

В этом рефакторинге мы будем использовать названия из **сокрытия делегирования**, а именно:

- *делегат* – конечный объект, который содержит функциональность, нужную клиенту;
- *сервер* – это объект, к которому клиент имеет непосредственный доступ.

Существует два вида проблем:

1. *Класс-сервер* ничего не делает сам по себе, создавая бесполезную сложность. В этом случае стоит задуматься, нужен ли этот класс вообще.
2. Каждый раз, когда в *делегате* появляется новая фича, для нее нужно создавать делегирующий метод в *классе-сервере*. Это бывает накладно при большом количестве изменений.

## Порядок рефакторинга

1. Создайте геттер для доступа к объекту *класса-делегата* из объекта *класса-сервера*.
2. Замените вызовы делегирующих методов *класса-сервера* прямыми вызовами методов *класса-делегата*.

## Анти-рефакторинг

### § Сокрытие делегирования

## Борется с запахом

### § Посредник



# ☒ Введение внешнего метода

Также известен как *Introduce Foreign Method*

## Проблема

Служебный класс не содержит метода, который вам нужен, при этом у вас нет возможности добавить метод в этот класс.

```
class Report {  
    //...  
    void sendReport() {  
        Date nextDay = new Date(previousEnd.getYear(),  
            previousEnd.getMonth(), previousEnd.getDate() + 1);  
        //...  
    }  
}
```

## Решение

Добавьте метод в клиентский класс и передавайте в него объект служебного класса в качестве аргумента.

```
class Report {  
    //...  
    void sendReport() {  
        Date newStart = nextDay(previousEnd);  
        //...  
    }  
  
    private static Date nextDay(Date arg) {  
        return new Date(arg.getYear(), arg.getMonth(), arg.getDate() + 1);  
    }  
}
```

## Причины рефакторинга

У вас есть код, который использует данные и методы определённого класса. Вы приходите к выводу, что этот код намного лучше будет смотреться и работать внутри нового метода в этом классе. Однако возможность добавить такой метод в класс у вас отсутствует (например, потому что класс находится в сторонней библиотеке).

Данный рефакторинг особенно выгоден в случаях, когда участок кода, который вы хотите перенести в метод, повторяется несколько раз в различных местах программы.

Так как вы передаёте объект служебного класса в параметры нового метода, у вас есть доступ ко всем его полям. Вы можете делать внутри этого метода практически все, что вам может потребоваться, как если бы метод был частью служебного класса.

## Достоинства

- Убирает дублирование кода. Если ваш участок кода повторяется в нескольких местах, вы можете заменить их вызовом метода. Это удобнее дублирования даже с учетом того, что внешний метод находится не там, где хотелось бы.

## Недостатки

- Причины того, почему метод служебного класса находится в клиентском классе, не всегда очевидны для того специалиста, который будет поддерживать код после вас. Если данный метод может быть использован и в других классах, имеет смысл создать обёртку над служебным классом, и поместить метод туда. То же самое имеет смысл сделать, если таких служебных методов несколько. В этом поможет рефакторинг [введение локального расширения](#).

## Порядок рефакторинга

1. Создайте новый метод в клиентском классе.
2. В этом методе создайте параметр, в который будет передаваться объект служебного класса. Если этот объект может быть получен из клиентского класса, параметр можно не создавать.
3. Извлеките волнующие вас участки кода в этот метод и замените их вызовами метода.
4. Обязательно оставьте в комментарии к этому методу метку *Foreign method* и призыв поместить этот метод в служебный класс, если такая возможность появится в дальнейшем. Это облегчит понимание того, почему этот метод находится в данном классе для тех, кто будет поддерживать программный продукт в будущем.

# **Родственные рефакторинги**

## **§ Введение локального расширения**

Вынести все расширенные методы в отдельный класс обёртку/наследник служебного класса.

## **Борется с запахом**

## **§ Неполнота библиотечного класса**

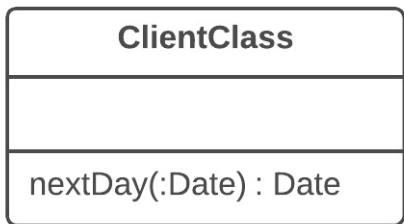


# ✖ Введение локального расширения

Также известен как *Introduce Local Extension*

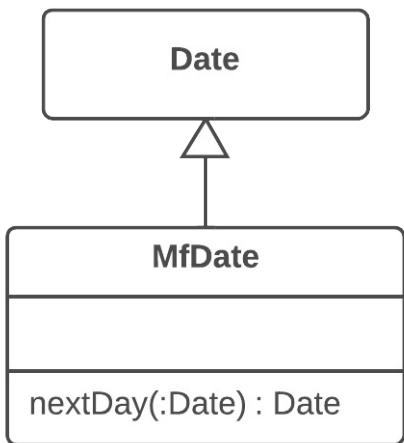
## Проблема

В служебном классе отсутствуют некоторые методы, которые вам нужны. При этом добавить их в этот класс вы не можете.



## Решение

Создайте новый класс, который бы содержал эти методы, и сделайте его наследником служебного класса, либо его обёрткой.



## Причины рефакторинга

В классе, который вы используете, нет нужных вам методов. Или ещё хуже – вы не можете их туда добавить (например, потому что классы находятся в сторонней библиотеке).

У вас есть два пути:

- Создать **подкласс** из интересующего класса, который будет содержать новые методы, и наследовать все остальное из родительского класса. Этот путь проще, но иногда бывает заблокирован в самом служебном классе с помощью директивы `final`.
- Создать **класс-обёртку**, который будет содержать все новые методы, а остальное делегировать связанному объекту служебного класса. Этот путь более трудоёмкий, так как вам нужно будет иметь в довесок не только код поддержания связи между обёрткой и служебным объектом, но и большое количество простых делегирующих методов, которые будут эмулировать публичный интерфейс служебного класса.

## Достоинства

- Помещая дополнительные методы в отдельный класс-расширение (обёртку или подкласс), вы не засоряете клиентские классы кодом, который им не принадлежит по смыслу. Этим повышается связность компонентов программы и возможность их повторного использования.

## Порядок рефакторинга

### 1. Создайте новый класс-расширение:

- либо сделайте его наследником служебного класса;
- либо, если вы решили делать обёртку, создайте в нем поле для хранения объекта служебного класса, к которому будет происходить делегирование. В этом случае нужно будет создать ещё и методы, которые повторяют публичные методы служебного класса и содержат простое делегирование к методам служебного объекта.

### 2. Создайте конструктор, использующий параметры конструктора служебного класса.

### 3. Кроме того, создайте альтернативный «конвертирующий» конструктор, который принимает в параметрах только объект оригинального класса. Это поможет в подстановке расширения вместо объектов оригинального класса.

### 4. Создайте в классе новые расширенные методы. Переместите в него внешние методы из других классов, либо удалите их, если расширение уже имеет такой функционал.

### 5. Замените использование служебного класса новым классом-расширением в тех местах, где нужна расширенная функциональность.

## Родственные рефакторинги

## **§ Введение внешнего метода**

Если вам не хватает только одного метода, иногда, проще поместить его в клиентский класс, в котором он вызывается, а служебный объект передавать в параметр этого метода.

## **Борется с запахом**

## **§ Неполнота библиотечного класса**



# Организация данных

Рефакторинги этой группы призваны облегчить работу с данными, заменив работу с примитивными типами богатыми функциональностью классами.

Кроме того, важным моментом является уменьшение связанность между классами, что улучшает переносимость классов и шансы их повторного использования.

## § Самоинкапсуляция поля

**Проблема:** Вы используете прямой доступ к приватным полям внутри класса.

**Решение:** Создайте геттер и сеттер для поля, и пользуйтесь для доступа к полю только ими.

## § Замена простого поля объектом

**Проблема:** В классе (или группе классов) есть поле простого типа. У этого поля есть своё поведение и связанные данные.

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

## § Замена значения ссылкой

**Проблема:** Есть много одинаковых экземпляров одного класса, которые можно заменить одним объектом.

**Решение:** Превратите одинаковые объекты в один объект-ссылку.

## § Замена ссылки значением

**Проблема:** У вас есть объект-ссылка, который слишком маленький и неизменяемый, чтобы оправдать сложности по управлению его жизненным циклом.

**Решение:** Превратите его в объект-значение.

## § Замена поля-массива объектом

**Проблема:** У вас есть массив, в котором хранятся разнотипные данные.

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

## § Дублирование видимых данных

**Проблема:** Данные предметной области программы хранятся в классах, отвечающих за пользовательский интерфейс (GUI).

**Решение:** Имеет смысл выделить данные предметной области в отдельные классы и, таким образом, обеспечить связь и синхронизацию между классом предметной области и GUI.

## § Замена односторонней связи двунаправленной

**Проблема:** У вас есть два класса, которым нужно использовать фичи друг друга, но между ними существует только односторонняя связь.

**Решение:** Добавьте недостающую связь в класс, в котором она отсутствует.

## § Замена двунаправленной связи односторонней

**Проблема:** У вас есть двухсторонняя связь между классами, но один из классов больше не использует фичи другого.

**Решение:** Уберите неиспользуемую связь.

## § Замена магического числа символьной константой

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

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

## § Инкапсуляция поля

**Проблема:** У вас есть публичное поле.

**Решение:** Сделайте поле приватным и создайте для него методы доступа.

## § Инкапсуляция коллекции

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

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

## § Замена кодирования типа классом

**Проблема:** В классе есть поле, содержащее кодирование типа. Значения этого типа не используются в условных операторах и не влияют на поведение программы.

**Решение:** Создайте новый класс и применяйте его объекты вместо значений закодированного типа.

## § Замена кодирования типа подклассами

**Проблема:** У вас есть закодированный тип, который непосредственно влияет на поведение программы (основываясь на значениях этого поля, в условных операторах выполняется различный код).

**Решение:** Для каждого значения закодированного типа, создайте подклассы. А затем, вынесите соответствующие поведения из исходного класса в эти подклассы. Управляющий код замените полиморфизмом.

### § Замена кодирования типа состоянием/стратегией

**Проблема:** У вас есть закодированный тип, который влияет на поведение, но вы не можете использовать подклассы, чтобы избавиться от него.

**Решение:** Замените кодирование типа объектом-состоянием. При необходимости заменить значение поля с кодированием типа, в него подставляется другой объект-состояние.

### § Замена подкласса полями

**Проблема:** У вас есть подклассы, которые отличаются только методами, возвращающими данные-константы.

**Решение:** Замените методы полями в родительском классе и удалите подклассы.



# Самоинкапсуляция поля

Также известен как *Self Encapsulate Field*

Самоинкапсуляция отличается от обычной инкапсуляции поля тем, что рефакторинг производится над приватным полем.

## Проблема

Вы используете прямой доступ к приватным полями внутри класса.

```
class Range {  
    private int low, high;  
  
    boolean includes(int arg) {  
        return arg >= low && arg <= high;  
    }  
}
```

## Решение

Создайте геттер и сеттер для поля, и пользуйтесь для доступа к полю только ими.

```
class Range {  
    private int low, high;  
  
    boolean includes(int arg) {  
        return arg >= getLow() && arg <= getHigh();  
    }  
  
    int getLow() {  
        return low;  
    }  
  
    int getHigh() {  
        return high;  
    }  
}
```

# Причины рефакторинга

Бывает так, что вам перестаёт хватать гибкости с прямым доступом к приватному полю внутри класса. Вы хотите иметь возможность инициализировать значение поля при первом запросе или производить какие-то операции над новыми значениями поля в момент присваивания, либо делать все это разными способами в подклассах.

## Достоинства

- *Непрямой доступ к полям* – это когда работа с полем происходит через методы доступа (геттеры и сеттеры). Этот подход отличается гораздо большей гибкостью, чем *прямой доступ к полям*.
  - Во-первых, вы можете осуществлять сложные операции при получении или установке данных в поле. *Ленивая инициализация, валидация значений в поле* – все это легко реализуемо внутри геттеров и сеттеров поля.
  - Во-вторых, что ещё важнее, вы можете переопределять геттеры и сеттеры в подклассах.
- Вы можете вообще не реализовывать сеттер для поля. Значение поля будет задаваться только в конструкторе, делая это поле неизменяемым для всего периода жизни объекта.

## Недостатки

- Когда используется *прямой доступ к полям*, код выглядит проще и нагляднее, хотя и теряет в гибкости.

## Порядок рефакторинга

1. Создайте геттер (и optionalный сеттер) для поля. Они должны быть защищёнными (`protected`) либо публичными (`public`).
2. Найдите все прямые обращения к полю и замените их вызовами геттера и сеттера.

## Родственные рефакторинги

### § Инкапсуляция поля

Скрываем публичные поля, создаём для них методы доступа.

## Помогает рефакторингу

- § Дублирование видимых данных
- § Замена кодирования типа подклассами
- § Замена кодирования типа состоянием/стратегией

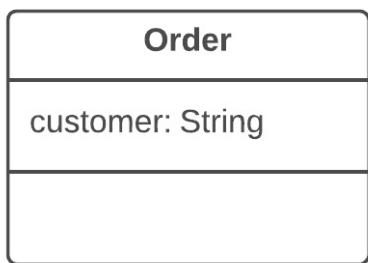


# ☒ Замена простого поля объектом

Также известен как *Replace Data Value with Object*

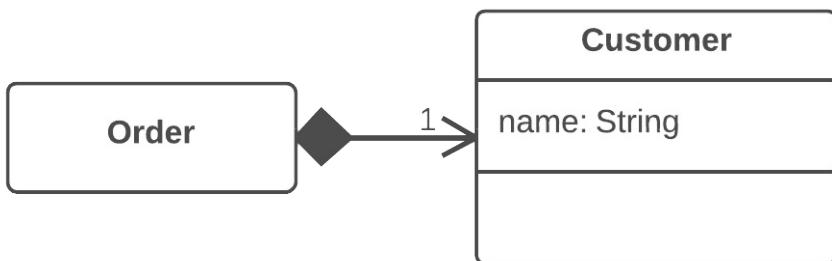
## Проблема

В классе (или группе классов) есть поле простого типа. У этого поля есть своё поведение и связанные данные.



## Решение

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



## Причины рефакторинга

В целом этот рефакторинг является частным случаем извлечения класса. Отличия заключаются в причинах, повлекших рефакторинг.

В извлечении класса мы имеем один класс, который отвечает за разные вещи, и хотим разделить его ответственность.

В замене простого поля объектом мы имеем поле примитивного типа (число, строка и пр.), которое с развитием программы перестало быть таким простым и обзавелось связанными данными и поведениями. С одной стороны, в наличии

таких полей нет ничего страшного, с другой – такое семейство-поляй-и-поведений может содержаться в нескольких классах одновременно, создавая много дублирующего кода.

Поэтому мы создаём для всего этого новый класс и перемещаем в него не только само поле, но и связанные с ним данные и поведения.

## Достоинства

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

## Порядок рефакторинга

Перед началом рефакторинга, посмотрите есть ли прямые обращения к полю внутри класса. Если да, примените самоинкапсуляцию поля, чтобы скрыть его в исходном классе.

1. Создайте новый класс и скопируйте в него ваше поле и его геттер. Кроме того, создайте конструктор, принимающий простое значение этого поля. В этом классе не будет сеттера, т.к. каждое новое значение поля, подаваемое в оригинальный класс, будет создавать новый объект-значение.
2. В исходном классе поменяйте тип поля на новый класс.
3. В его геттере в исходном классе обращайтесь к геттеру связанного объекта.
4. В сеттере создайте новый объект-значение. Возможно, вам потребуется также создать новый объект в конструкторе, если ранее там задавались какие-то начальные значения для поля.

## Последующие шаги

После этого рефакторинга иногда имеет смысл применить замену значения ссылкой над полем, содержащим объект. Это позволяет вместо хранения десятков объектов для одного и того же значения поля хранить ссылку на один объект, соответствующий этому значению.

Чаще всего такое решение потребуется в случаях, когда вы хотите иметь один объект, отвечающий за один реальный объект из жизни (например, пользователи, заказы, документы и так далее). В тоже время такой подход будет лишним для объектов дат, денег, диапазонов и т.п.

## Родственные рефакторинги

- § Замена параметров объектом
- § Замена поля-массива объектом
- § Замена метода объектом методов

| Делает то же, но с кодом метода.

## **Борется с запахом**

- § Дублирование кода

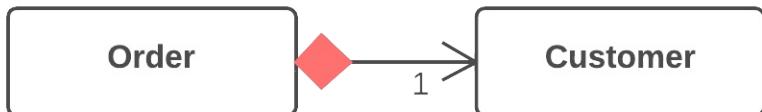


# ☒ Замена значения ссылкой

Также известен как *Change Value to Reference*

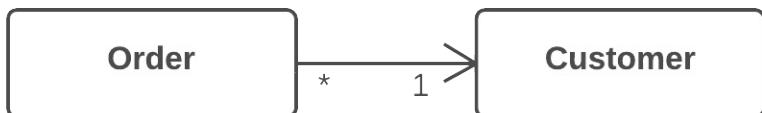
## Проблема

Есть много одинаковых экземпляров одного класса, которые можно заменить одним объектом.



## Решение

Превратите одинаковые объекты в один объект-ссылку.



## Причины рефакторинга

Во многих системах объекты можно разделить на две категории – объекты-значения и объекты-ссылки.

- **Объекты-ссылки** – это когда одному объекту реального мира соответствует только один объект программы. Объектами-ссылками обычно становятся объекты пользователей, заказов, товаров и пр.
- **Объекты-значения** – одному объекту реального мира соответствует множество объектов в программе. Такими объектами могут быть даты, телефонные номера, адреса, цвет и другие.

Выбор между ссылкой и значением не всегда очевиден. Иногда вначале есть простое значение с небольшим объёмом неизменяемых данных. Затем возникает необходимость добавить изменяемые данные и обеспечить передачу этих изменений при всех обращениях к объекту. Тогда появляется необходимость превратить его в объект-ссылку.

## Достоинства

- Один объект хранит в себе всю последнюю информацию об определенной сущности. Если в этот объект вносятся изменения в одной части программы, они тут же становятся доступными из других частей программы, использующих этот объект.

## Недостатки

- Объекты-ссылки гораздо сложнее в реализации.

## Порядок рефакторинга

1. Используйте замену конструктора фабричным методом над классом, который должен порождать объекты-ссылки.
2. Определите, какой объект будет ответственным за предоставление доступа к объектам-ссылкам. Вместо создания нового объекта, когда он нужен, вам теперь нужно получать его из какого-то объекта-хранилища или статического поля-словаря.
3. Определите, будут ли объекты-ссылки создаваться заранее или динамически по мере необходимости. Если объекты создаются предварительно, необходимо обеспечить их загрузку перед использованием.
4. Измените фабричный метод так, чтобы он возвращал объект-ссылку. Если объекты создаются заранее, необходимо решить, как обрабатывать ошибки при запросе несуществующего объекта. Также вам может понадобиться применить к фабрике переименование метода для информации о том, что метод возвращает только существующие объекты.

## Анти-рефакторинг

### § Замена ссылки значением

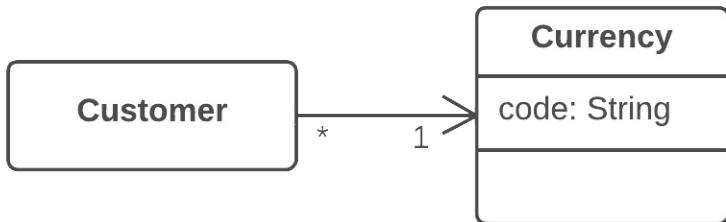


# ☒ Замена ссылки значением

Также известен как *Change Reference to Value*

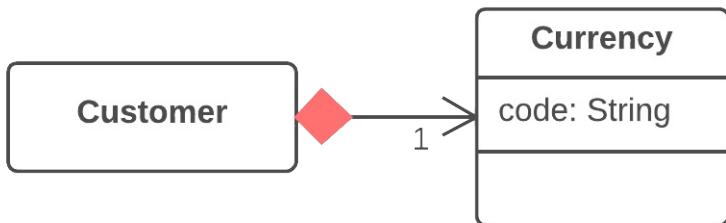
## Проблема

У вас есть объект-ссылка, который слишком маленький и неизменяемый, чтобы оправдать сложности по управлению его жизненным циклом.



## Решение

Превратите его в объект-значение.



## Причины рефакторинга

Побудить к переходу от ссылки к значению могут возникшие при работе с объектом-ссылкой неудобства.

Объектами-ссылками необходимо некоторым образом управлять:

- всегда приходится запрашивать у объекта-хранилища нужный объект;
- ссылки в памяти тоже могут оказаться неудобными в работе;
- работать с объектами-ссылками, в отличие от объектов-значений, особенно сложно в распределённых и параллельных системах.

Кроме того, объекты-значения будут особенно полезны, если вам больше подходит неизменяемость объектов, чем возможность изменения их состояния во

время жизни объекта.

## Достоинства

- Важное свойство объектов-значений заключается в том, что они должны быть неизменяемыми. При каждом запросе, возвращающем значение одного из них, должен получаться одинаковый результат. Если это так, то не возникает проблем при наличии многих объектов, представляющих одну и ту же вещь.
- Объекты-значения гораздо проще в реализации.

## Недостатки

- Если значение изменяемое, вам необходимо обеспечить, чтобы при изменении любого из объектов обновлялись значения во всех остальных, которые представляют ту же самую сущность. Это настолько обременительно, что проще для этого создать объект-ссылку.

## Порядок рефакторинга

1. Обеспечьте неизменяемость объекта. Объект не должен иметь сеттеров или других методов, меняющих его состояние и данные (в этом может помочь **удаление сеттера**). Единственное место, где полям объекта-значения присваиваются какие-то данные, должен быть конструктор.
2. Создайте метод сравнения для того, чтобы иметь возможность сравнить два объекта-значения.
3. Проверьте, возможно ли удалить фабричный метод и сделать конструктор объекта публичным.

## Анти-рефакторинг

### § Замена значения ссылкой



## ☒ Замена поля-массива объектом

Также известен как *Replace Array with Object*

Этот рефакторинг является особым случаем замены простого поля объектом.

## Проблема

У вас есть массив, в котором хранятся разнотипные данные.

```
String[] row = new String[2];
row[0] = "Liverpool";
row[1] = "15";
```

## Решение

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

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

## Причины рефакторинга

Массивы – отличный инструмент для хранения однотипных данных и коллекций. Но ждите беды, если вы используете массив в качестве банковских ячеек, например, храните в ячейке №1 – имя пользователя, а в ячейке №14 – его адрес. Такой подход не только может привести к фатальным последствиям, когда кто-то положит что-то не в ту ячейку, но также требует затраты огромного количества времени на запоминание того, где какие данные хранятся.

## Достоинства

- В образовавшийся класс можно переместить все связанные поведения, которые раньше хранились в основном классе или в других местах.
- Поля класса гораздо проще документировать, чем ячейки массива.

## Порядок рефакторинга

1. Создайте новый класс, который будет содержать данные из массива. Поместите в него сам массив как публичное поле.
2. Создайте поле для хранения объекта этого класса в исходном классе. Не забудьте создать также сам объект в том месте, где вы инициировали массив данных.
3. В новом классе один за другим создавайте методы доступа для всех элементов массива. Давайте им понятные названия, которые отражают суть того, что они хранят. В тоже время изменяйте каждый случай использования ячейки массива в основном коде соответствующими методами доступа к этой ячейке.
4. Когда методы доступа будут созданы для всех ячеек, сделайте массив приватным.
5. Для каждого элемента массива создайте приватное поле в классе, после чего измените методы доступа так, чтобы они использовали это поле вместе массива.
6. Когда все данные будут перемещены, удалите массив.

## Родственные рефакторинги

### § Замена простого поля объектом

## Борется с запахом

### § Одержанность элементарными типами

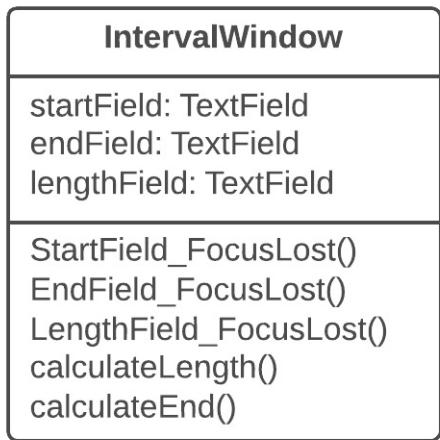


# ✖ Дублирование видимых данных

Также известен как *Duplicate Observed Data*

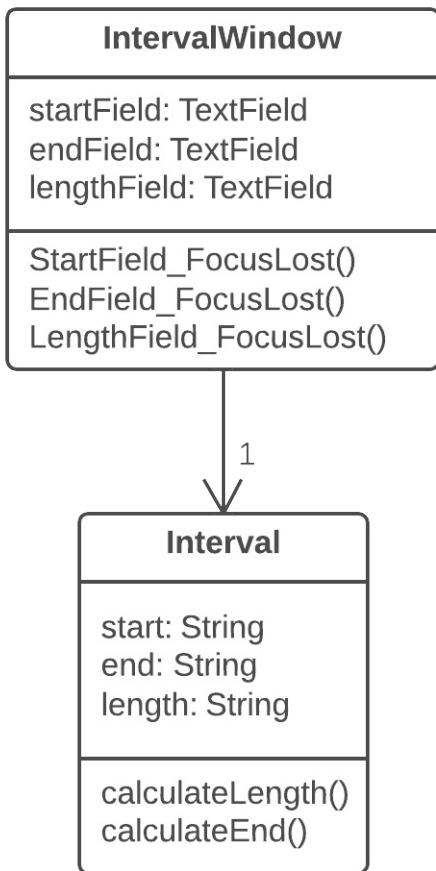
## Проблема

Данные предметной области программы хранятся в классах, отвечающих за пользовательский интерфейс (GUI).



## Решение

Имеет смысл выделить данные предметной области в отдельные классы и, таким образом, обеспечить связь и синхронизацию между классом предметной области и GUI.



## Причины рефакторинга

Вы хотите иметь несколько видов интерфейса для одних и тех же данных (например, у вас есть приложение не только для десктопа, но также для телефонов и планшетов). В этом случае вам будет очень сложно избежать большого количества ошибок и дублирования кода, если вы не разделите GUI и предметную область.

## Достоинства

- Вы разделяете ответственность между классами бизнес-логики и представления (*принцип единственной обязанности*), что упрощает читабельность и понимание программы в целом.
- Если потребуется добавить новый вид интерфейса, вам нужно будет создать новые классы представления, при этом код бизнес-логики трогать нет никакой нужды (*принцип открытости/закрытости*).
- Над бизнес-логикой и пользовательскими интерфейсами теперь могут работать разные люди.

# Когда нельзя применить

- Этот рефакторинг, который в классическом исполнении производится с введением паттерна **Наблюдатель**, малоприменим для веб-приложений, где все классы пересоздаются между запросами к веб-серверу.
- Тем не менее, общий принцип извлечения бизнес-логики в отдельные классы имеет смысл, в том числе, и для веб-приложений. Но реализуется он при помощи других рефакторингов, исходя из дизайна вашей системы.

## Порядок рефакторинга

1. Необходимо скрыть прямой доступ к данным предметной области в *классе GUI*, для чего лучше всего использовать **самоинкапсуляцию поля**. Таким образом, вы создадите геттеры и сеттеры к этим данным.
2. В обработчиках событий *класса GUI* используйте сеттеры для установки новых значений полей. Это даст возможность передавать новые значения в связанный *объект предметной области*.
3. Создайте класс предметной области и скопируйте в него необходимые поля из *класса GUI*. Для всех этих полей создайте геттеры и сеттеры.
4. Примените паттерн Наблюдатель к этим двум классам:
  - В *классе предметной области* создайте массив для хранения объектов наблюдателей (*объектов GUI*), а также методы их регистрации, удаления и оповещения.
  - В *классе GUI* создайте поле для хранения ссылки на *объект предметной области*, а также метод `update()`, который будет реагировать на изменения в этом объекте и обновлять значения полей в *классе GUI*. Обратите внимание, в методе обновления значения должны устанавливаться напрямую, чтобы избежать рекурсии.
  - В конструкторе *класса GUI* создайте экземпляр *класса предметной области* и сохраните его в созданном поле. Зарегистрируйте *объект GUI* как наблюдатель в *объекте предметной области*.
  - В сеттерах полей *класса предметной области* вызывайте метод оповещения наблюдателя (т.е. метод обновления в *классе GUI*), чтобы передать новые значения в пользовательский интерфейс.
  - Измените сеттеры полей *класса GUI* так, чтобы они теперь устанавливали новые значения в *объекте предметной области*, причём напрямую. Будьте внимательны, если значения будут устанавливаться через сеттер *класса предметной области*, это приведёт к бесконечной рекурсии.

# Реализует паттерн проектирования

§ Наблюдатель



## Борется с запахом

§ Большой класс





# ✖ Замена односторонней связи двунаправленной

Также известен как *Change Unidirectional Association to Bidirectional*

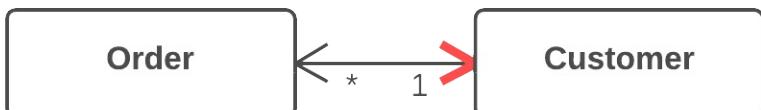
## Проблема

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



## Решение

Добавьте недостающую связь в класс, в котором она отсутствует.



## 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.

## Порядок рефакторинга

1. Добавьте поле, которое будет содержать обратную связь.
2. Решите, какой класс будет «управляющим». Этот класс должен содержать методы, которые создают или обновляют связь при добавлении или изменении элементов, устанавливая связь в своём классе, а также вызывая служебные методы установки связи в связываемом объекте.
3. Создайте служебный метод установки связи в «не управляющем классе». Этот метод должен заполнять поле со связью тем, что ему подают в параметрах. Назовите этот метод так, чтобы было очевидно, что его не стоит использовать в других целях.
4. Если старые методы управления односторонней связью находились в «управляющем классе», дополните их вызовами служебных методов из связываемого объекта.
5. Если старые методы управления связью находились в «не управляющем классе», создайте управляющие методы в «управляющем классе», вызывайте их и делегируйте им выполнение.

## Анти-рефакторинг

### § Замена двунаправленной связи односторонней

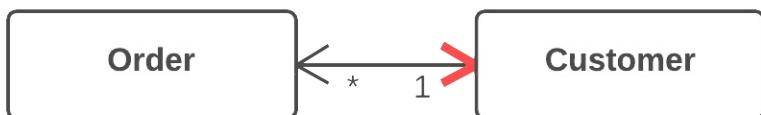


# ☒ Замена двунаправленной связи односторонней

Также известен как *Change Bidirectional Association to Unidirectional*

## Проблема

У вас есть двухсторонняя связь между классами, но один из классов больше не использует фичи другого.



## Решение

Уберите неиспользуемую связь.



## Причины рефакторинга

Двустороннюю связь, как правило, сложнее поддерживать, чем одностороннюю. Такая связь требует наличия дополнительного кода, который бы отслеживал корректное создание и удаление связанных объектов, что приводит к усложнению программы.

Кроме того, неправильно реализованная двусторонняя связь может приводить к проблемам с «очисткой мусора», т.е. освобождения памяти, занятой неиспользуемыми объектами.

Приведём пример: «сборщик мусора» удаляет из памяти объекты, на которые уже не осталось ссылок из других объектов программы. Допустим, была создана, использована, после чего заброшена пара объектов `Пользователь - Заказ`. Однако эти объекты не будут очищены из памяти, так как они все ещё содержат ссылки друг на друга. Стоит, однако, заметить, что эта проблема теряет актуальность с течением времени. С ней уже научились бороться многие современные версии языков программирования, которые автоматически обнаруживают неиспользуемые связки объектов и успешно очищают их из памяти.

Кроме того, существует проблема тесной зависимости между классами. Двусторонняя связь предполагает, что два класса должны знать друг о друге, а значит, такие классы невозможно использовать отдельно друг от друга. Наличие множества таких двусторонних связей приводит к тому, что части программы становятся слишком тесно связанными, и любое изменение в одном из компонентов приводит к необходимости менять другие компоненты программы.

## Достоинства

- Упрощает код класса, которому не нужна связь. Меньше кода – проще поддержка.
- Уменьшает зависимость между классами. Независимые классы проще поддерживать, т.к. изменения в них затрагивают только эти классы.

## Порядок рефакторинга

1. Убедитесь, что один из следующих пунктов справедлив в отношении ваших классов:
  - связь не используется вообще;
  - есть другой способ получить связанный объект, например, используя запрос к базе данных;
  - связанный объект может быть передан как аргумент в методы, которые используют его.
2. Избавьтесь от использование поля, содержащего связь с другим объектом. Проще всего заменить такую связь передачей нужного объекта в параметрах метода.
3. Удалите код присваивания связанного объекта полю.
4. Удалите неиспользуемое теперь поле.

## Анти-рефакторинг

### § Замена односторонней связи двунаправленной

## Борется с запахом

### § Неуместная близость



# Замена магического числа символической константой

Также известен как *Replace Magic Number with Symbolic Constant*

## Проблема

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

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

## Решение

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

```
static final double GRAVITATIONAL_CONSTANT = 9.81;  
  
double potentialEnergy(double mass, double height) {  
    return mass * height * GRAVITATIONAL_CONSTANT;  
}
```

## Причины рефакторинга

Магические числа – это числовые значения, встречающиеся в коде, но при этом неочевидно, что они означают. Данный антипаттерн затрудняет понимание программы и усложняет её рефакторинг.

Дополнительные сложности возникают, когда нужно поменять определённое магическое число. Это нельзя сделать автозаменой, т.к. одно и то же число может использоваться для разных целей, а значит, вам нужно будет проверять каждый участок кода, где используется это число.

## Достоинства

- Символьная константа может служить живой документацией смысла значения, которое в ней хранится.
- Значение константы намного проще заменить, чем искать нужное число по всему коду, при этом рискуя заменить такое же число, которое в данном

конкретном случае использовалось для других целей.

- Убирает дублирование использования числа или строки по всему коду. Это особенно актуально, если значение является сложным и длинным (например, `-14159`, `0xCAFEBAE`).

## Полезные факты

**Не все числа являются магическими.**

Если предназначения чисел очевидны, их не надо заменять константами, классический пример:

```
for (i = 0; i < count; i++) ... ;
```

### Альтернативы

1. Иногда, магическое число можно заменить вызовом метода. Например, если у вас есть магическое число, обозначающее количество элементов коллекции, вам не обязательно использовать его для проверок последнего элемента коллекции. Вместо этого можно использовать встроенный метод получения длины коллекции.
2. Магические числа могут быть использованы для реализации кодирования типа. Например, у вас есть два типа пользователей, и чтобы обозначить их, у вас есть числовое поле в классе, в котором для администраторов хранится число `1`, а для простых пользователей – число `2`.

В этом случае имеет смысл использовать один из рефакторингов избавления от кодирования типа:

- замена кодирования типа классом
- замена кодирования типа подклассами
- замена кодирования типа состоянием/стратегией

## Порядок рефакторинга

1. Объявите константу и присвойте ей значение магического числа.
2. Найдите все упоминания магического числа.
3. Для всех найденных чисел проверьте, согласуется ли это магическое число с предназначением константы. Если да, замените его вашей константой. Эта проверка важна, т.к. одно и тоже число может означать совершенно разные вещи (в этом случае, они должны быть заменены разными константами).



# ✂ Инкапсуляция поля

Также известен как *Encapsulate Field*

## Проблема

У вас есть публичное поле.

```
class Person {  
    public String name;  
}
```

## Решение

Сделайте поле приватным и создайте для него методы доступа.

```
class Person {  
  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String arg) {  
        name = arg;  
    }  
}
```

## Причины рефакторинга

Одним из столпов объектного программирования является *Инкапсуляция* или возможность скрытия данных объекта. Иначе все данные объектов были бы публичными, и другие объекты могли бы получать и модифицировать данные вашего объекта без его ведома. При этом отделяются данные от поведений, связанных с этими данными, ухудшается модульность частей программы и усложняется её поддержка.

## Достоинства

- Если данные и поведения какого-то компонента тесно связаны между собой и находятся в одном месте в коде, вам гораздо проще поддерживать и развивать этот компонент.
- Кроме того, вы можете производить какие-то сложные операции, связанные с доступом к полям объекта.

## Когда нельзя применить

- Встречаются случаи, когда инкапсуляция полей нежелательна из соображений, связанных с повышением производительности. Эти случаи очень редки, но иногда этот момент бывает очень важным.

Например, у вас есть графический редактор, в котором есть объекты, имеющие координаты `x` и `y`. Эти поля вряд ли будут меняться в будущем. К тому же, в программе участвует очень много различных объектов, в которых присутствуют эти поля. Поэтому обращение напрямую к полям координат экономит значительную часть процессорного времени, которое иначе затрачивалось бы на вызовы методов доступа.

Как иллюстрация этого исключения, существует класс `Point` в Java, все поля которого являются публичными.

## Порядок рефакторинга

1. Создайте геттер и сеттер для поля.
2. Найдите все обращения к полю. Замените получение значения из поля геттером, а установку новых значений в поле – сеттером.
3. После того, как все обращения к полям заменены, сделайте поле приватным.

## Последующие шаги

«Инкапсуляция поля» является всего лишь первым шагом к сближению данных и поведений над этими данными. После того, как вы создали простые методы доступа к полям, стоит ещё раз проверить места, где эти методы вызываются. Вполне возможно, код из этих участков уместнее смотрелся бы в самих методах доступа.

## Родственные рефакторинги

### § Самоинкапсуляция поля

Вместо прямого доступа к приватным полям, создаём методы доступа к этим полям.

# **Борется с запахом**

§ Класс данных



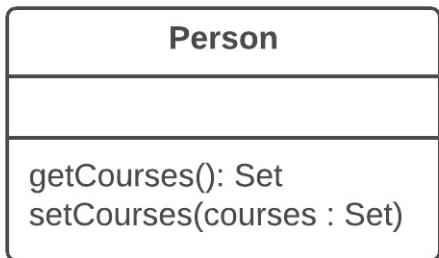


# Инкапсуляция коллекции

Также известен как *Encapsulate Collection*

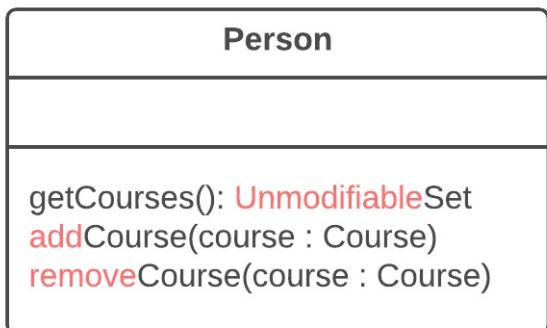
## Проблема

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



## Решение

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



## Причины рефакторинга

В классе есть поле, содержащее коллекцию объектов. Эта коллекция может быть массивом, списком, множеством или вектором. Для работы с этой коллекцией создан обычный геттер и сеттер.

Однако коллекции должны использовать протокол, несколько отличный от того, который используется другими типами данных. Метод получения не должен возвращать сам объект коллекции, потому что это позволило бы клиентам

изменять содержимое коллекции без ведома владеющего ею класса. Кроме того, это чрезмерно раскрывало бы клиентам строение внутренних структур данных объекта. Метод получения элементов коллекции должен возвращать такое значение, которое не позволяло бы изменять коллекцию и не раскрывало бы лишних данных о её структуре.

Кроме того, не должно быть метода, присваивающего коллекции значение; вместо этого должны быть операции для добавления и удаления элементов. Благодаря этому объект-владелец получает контроль над добавлением и удалением элементов коллекции.

Такой протокол осуществляет корректную инкапсуляцию коллекции, что, в итоге, уменьшает связанность между владеющим ею классом и клиентским кодом.

## Достоинства

- Поле коллекции инкапсулировано внутри класса. При вызове геттера возвращается копия коллекции, что исключает возможность случайного изменения или перетирания элементов коллекции, без ведома того объекта, в котором она хранится.
- В случае если элементы коллекции содержаться внутри примитивного типа, вроде массива, вы создаёте более удобные методы работы с коллекцией.
- Если элементы коллекции содержаться внутри непримитивного контейнера (стандартного класса коллекций), инкапсулировав коллекцию, вы можете ограничить доступ к нежелательным стандартным методам коллекции (например, ограничив добавление новых элементов).

## Порядок рефакторинга

1. Создайте методы для добавления и удаления элементов коллекции. Они должны принимать элементы коллекции в параметрах.
2. Присвойте полю пустую коллекцию в качестве начального значения, если это ещё не делается в конструкторе класса.
3. Найдите вызовы сеттера поля коллекции. Измените сеттер так, чтобы он использовал операции добавления и удаления элементов, или сделайте так, чтобы эти операции вызывали клиентский код.

Обратите внимание, что сеттеры могут быть использованы только для того, чтобы заменить все элементы коллекции на другие. Исходя из этого, возможно, имеет смысл изменить название сеттера на `replace`.

4. Найдите все вызовы геттера коллекции, после которых происходит изменение коллекции. Поменяйте этот код так, чтобы там использовались ваши новые

методы добавления и удаления элементов коллекции.

5. Измените геттер, чтобы он возвращал представление коллекции, доступное только для чтения.
6. Обследуйте клиентский код, использующий коллекцию, в поисках кода, который бы лучше смотрелся внутри самого класса коллекции.

## Борется с запахом

§ Класс данных





# Замена кодирования типа классом

Также известен как *Replace Type Code with Class*

**Что такое кодирование типа?** 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.

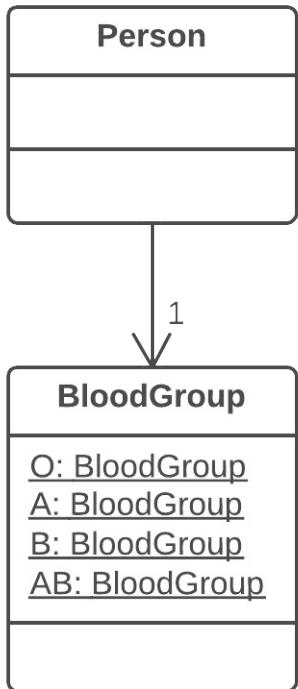
## Проблема

В классе есть поле, содержащее кодирование типа. Значения этого типа не используются в условных операторах и не влияют на поведение программы.

Person
O: int
A: int
B: int
AB: int
bloodgroup: int

## Решение

Создайте новый класс и применяйте его объекты вместо значений закодированного типа.



## Причины рефакторинга

Одной из самых частых причин появления кодирования типа является работа с базой данных, в полях которой какая-то сложная концепция кодируется каким-то числом или строкой.

Например, у вас есть класс `Пользователь` с полем `роль_пользователя`, в которой содержится информация о доступе каждого пользователя, будь то администратор, редактор или обычный пользователь. Причём эта информация кодируется в поле как символы `A`, `E`, и `U` соответственно.

Недостатками этого подхода является то, что в сеттерах поля зачастую нет никаких проверок на то, какое значение туда приходит, что может привести к большим проблемам, если кто-то отправит в эти поля какие-то другие значения.

Проблема также усугубляется тем, что для таких полей невозможно сделать проверку типов. В них можно отправить любое число или строку, что, безусловно, останется без внимания проверки типов вашей IDE и даже даст возможность программе запуститься.

## Достоинства

- Мы хотим превратить наборы значений примитивных типов, коими являются закодированные типы, в стройные классы, которые бы обладали всеми мощными свойствами ООП.

- Заменив кодирование типа классами, мы обеспечим возможность контроля и проверки типов значений (type hinting), передаваемых в методы и поля на уровне языка программирования.  
Например, если раньше, при передаче значения в метод компилятор не видел разницы между вашей числовой константой и каким-то произвольным числом, то теперь при передаче данных, отличающихся чем-то от указанного класса типа, вы получите сообщения об ошибке ещё внутри вашей IDE.
- Таким образом, мы сделаем возможным перенос кода в сами классы типа. Если вам нужно было проводить какие-то сложные манипуляции со значениями типа по всей программе, теперь этот код сможет «жить» внутри одного или нескольких классов типа.

## Когда нельзя применить

- Если значения закодированного типа используются внутри управляющих структур (`if`, `switch` и др.), и управляют каким-то поведением класса, вам следует прибегнуть к одному из двух других рефакторингов устранения кодирования типа:
  - замена кодирования типа подклассами
  - замена кодирования типа состоянием/стратегией

## Порядок рефакторинга

1. Создайте новый класс и дайте ему понятное название, соответствующее предназначению закодированного типа. Мы будем называть его *класс типа*.
2. В *классе типа* скопируйте поле, содержащее кодирование типа, и сделайте его приватным. Кроме того, создайте для этого поля геттер. Устанавливаться значение в это поле будет только из конструктора.
3. Для каждого значения закодированного типа, создайте статический метод в *классе типа*. Он будет создавать новый объект *класса типа*, соответствующий этому значению закодированного типа.
4. В исходном классе, замените тип закодированного поля на *класс типа*. Создавайте новый объект этого типа в конструкторе, а также в сеттере поля. Измените геттер поля так, чтобы он вызывал геттер *класса типа*.
5. Замените любые упоминания значений закодированного типа вызовами соответствующих статических методов *класса типа*.
6. Удалите константы закодированного типа из исходного класса.

# **Родственные рефакторинги**

- § Замена кодирования типа подклассами
- § Замена кодирования типа состоянием/стратегией

## **Борется с запахом**

- § Одержанность элементарными типами



# ☒ Замена кодирования типа подклассами

Также известен как *Replace Type Code with Subclasses*

**Что такое кодирование типа?** 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.

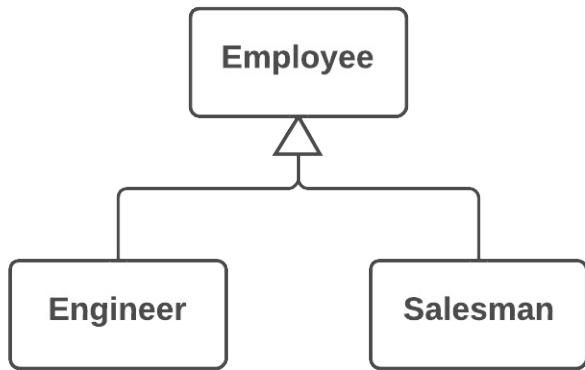
## Проблема

У вас есть закодированный тип, который непосредственно влияет на поведение программы (основываясь на значениях этого поля, в условных операторах выполняется различный код).



## Решение

Для каждого значения закодированного типа, создайте подклассы. А затем, вынесите соответствующие поведения из исходного класса в эти подклассы. Управляющий код замените полиморфизмом.



## Причины рефакторинга

Данный рефакторинг является более сложным случаем замены кодирования типа классом.

Как и в первом рефакторинге, у вас есть какой-то набор простых значений, которые составляют все доступные значения для какого-то поля. Хотя эти значения зачастую заданы как константы и имеют понятные имена, их использование чревато появлением ошибок проверки типов, так как, в сущности, они всё-равно являются значениями примитивных типов. Например, у вас есть метод, принимающий в параметрах одно из таких значений. В определённый момент, вместо константы `USER_TYPE_ADMIN` со значением `"ADMIN"`, в метод придёт та же строка, но в нижнем регистре `"admin"`, что приведёт к выполнению чего-то другого, нежели планируемое автором поведение.

В данном рефакторинге мы имеем дело с управляемым кодом, таким как условные операторы `if`, `switch` или `?:`. Другими словами, внутри условий этих операторов используются поля с закодированным значением (напр.

`$user->type == self::USER_TYPE_ADMIN`). Если бы мы применяли здесь замену кодирования типа классом, то все эти управляющие конструкции следовало бы тоже перенести в класс, отвечающих за тип данных. Это, в конечном итоге, сделало бы класс типа очень похожим на исходный класс, и содержащим те же проблемы.

## Достоинства

- Удаление управляющего кода. Вместо большого `switch` в исходном классе, вы переносите код в соответствующие подклассы. Это улучшает *разделение ответственности между классами* и упрощает читабельность программы в целом.

- Если вам потребуется добавить новое значение закодированного типа, все что нужно будет сделать – это добавить новый подкласс, не трогая существующий код (*принцип открытости/закрытости*).
- Заменив кодирование типа классами, мы обеспечим возможность контроля и проверки типов значений (type hinting), передаваемых в методы и поля на уровне языка программирования. Чего не сделаешь при помощи простых численных или строковых значений, содержащихся в закодированном типе.

## Когда нельзя применить

- Этот рефакторинг невозможно применить, если у вас уже есть какая-то иерархия классов. Вы не можете создать двойную иерархию при помощи наследования в ООП. Тем не менее, замену кодирования типа можно осуществить, используя композицию вместо наследования. Для этого используйте замену кодирования типа состоянием/стратегией.
- Если значения закодированного поля может поменяться после того, как объект был создан. При этом нам бы пришлось как-то заменить класс самого объекта на лету, а это не возможно. Тем не менее, альтернативой и здесь является применение замены кодирования типа состоянием/стратегией.

## Порядок рефакторинга

1. Используйте самоинкапсуляцию поля для создания геттера для поля, которое содержит кодирование типа.
2. Сделайте конструктор суперкласса приватным. Создайте статический фабричный метод с теми же параметрами, что и конструктор суперкласса. Он обязательно должен содержать параметр, который будет принимать стартовые значения закодированного типа. В зависимости от этого параметра, фабричный метод будет создавать объекты различных подклассов. Для этого в его коде придётся создать большой условный оператор, но, по крайней мере, он будет единственным, который действительно необходим, во всем остальном, в дальнейшем, могут позаботиться подклассы и полиморфизм.
3. Для каждого значения кодированного типа, создайте свой подкласс. В нем переопределите геттер закодированного поля так, чтобы он возвращал соответствующее значение закодированного типа.
4. Удалите поле с закодированным типом из суперкласса, его геттер сделайте абстрактным.
5. Теперь, когда у вас появились подклассы, можете начинать перемещать поля и методы из суперкласса в соответствующие подклассы (при помощи спуска поля и спуска метода).

5. Когда все что можно перемещено, используйте замену условных операторов полиморфизмом, чтобы окончательно избавиться от условных операторов, использующий закодированный тип.

## Анти-рефакторинг

- § Замена подкласса полями

## Родственные рефакторинги

- § Замена кодирования типа классом
- § Замена кодирования типа состоянием/стратегией

## Борется с запахом

- § Одержанность элементарными типами



# Замена кодирования типа состоянием/стратегией

Также известен как *Replace Type Code with State/Strategy*

**Что такое кодирование типа?** 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.

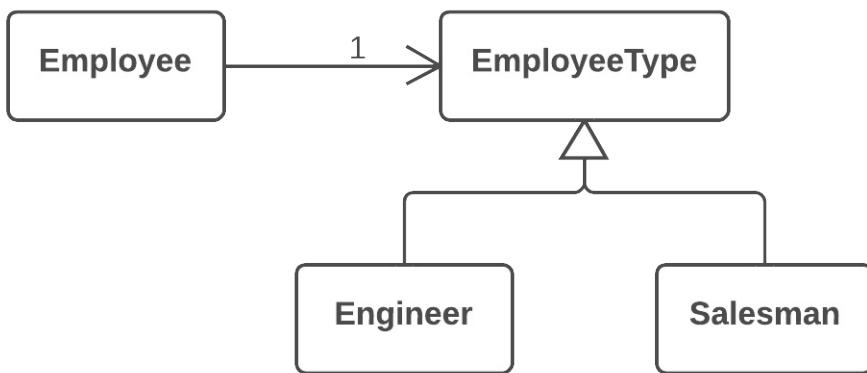
## Проблема

У вас есть закодированный тип, который влияет на поведение, но вы не можете использовать подклассы, чтобы избавиться от него.



## Решение

Замените кодирование типа объектом-состоянием. При необходимости заменить значение поля с кодированием типа, в него подставляется другой объект-состояние.



## Причины рефакторинга

У нас есть кодирование типа, и оно влияет на поведение класса, поэтому мы не можем заменить кодирования типа классом.

Кодирование типа влияет на поведение класса, но мы не можем создавать подклассы для закодированного типа ввиду наличия существующей иерархии классов или других причин. А значит, не можем применить замену кодирования типа подклассами.

## Достоинства

- Данный рефакторинг предоставляет выход из ситуации, когда поле с закодированным типом меняет своё значение в процессе жизни объекта. В этом случае, замена значения производится путём замены объекта-состояния, на который ссылается исходный класс.
- Если вам потребуется добавить новое значение закодированного типа, все что придётся сделать – это добавить новый подкласс-состояние, не трогая существующий код (*принцип открытости/закрытости*).

## Недостатки

- Если у вас есть простой случай кодирования типа, но вы все равно применяете данный рефакторинг, то у вас появится много лишних классов.

## Полезные факты

Данный рефакторинг может использовать реализацию одного из двух паттернов проектирования – *Состояния* либо *Стратегии*. Реализация этого рефакторинга

остаётся той же, вне зависимости от того, какой из паттернов вы выберете. Тем не менее, какой паттерн стоит выбрать в вашей ситуации?

Вам подойдёт Стратегия, если вы пытаетесь разделить условный оператор, управляющий выбором того или иного алгоритма.

Однако если каждое значение закодированного типа отвечает не просто за альтернативную версию алгоритма, а за целое состояние класса, значение полей и множество других действий, вам больше подойдёт Состояние.

## Порядок рефакторинга

1. Используйте самоинкапсуляцию поля для создания геттера для поля, которое содержит кодирование типа.
2. Создайте новый класс и дайте ему понятное название, соответствующее предназначению закодированного типа. Этот класс будет играть роль *состояния* (или *стратегии*). Создайте в нем абстрактный геттер закодированного поля.
3. Создайте подклассы класса-состояния для каждого значения закодированного типа. В каждом подклассе переопределите геттер закодированного поля так, чтобы он возвращал соответствующее значение закодированного типа.
4. В абстрактном классе состояния, создайте статический фабричный метод, принимающий в параметре значение закодированного типа. В зависимости от этого параметра, фабричные методы будут создавать объекты различных состояний. Для этого в его коде придётся создать большой условный оператор, но он будет единственным по завершению рефакторинга.
5. В исходном классе, поменяйте тип закодированного поля на класс-состояние. В сеттере этого поля, вызывайте фабричный метод состояния для получения новых объектов состояний.
6. Теперь, можете начинать перемещать поля и методы из суперкласса в соответствующие подклассы-состояния (при помощи спуска поля и спуска метода).
7. Когда все что можно перемещено, используйте замену условных операторов полиморфизмом, чтобы окончательно избавиться от условных операторов, использующий закодированный тип.

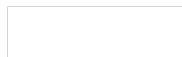
## Родственные рефакторинги

§ Замена кодирования типа классом

§ Замена кодирования типа подклассами

# Реализует паттерн проектирования

§ Состояние



§ Стратегия



## Борется с запахом

§ Одержанность элементарными типами



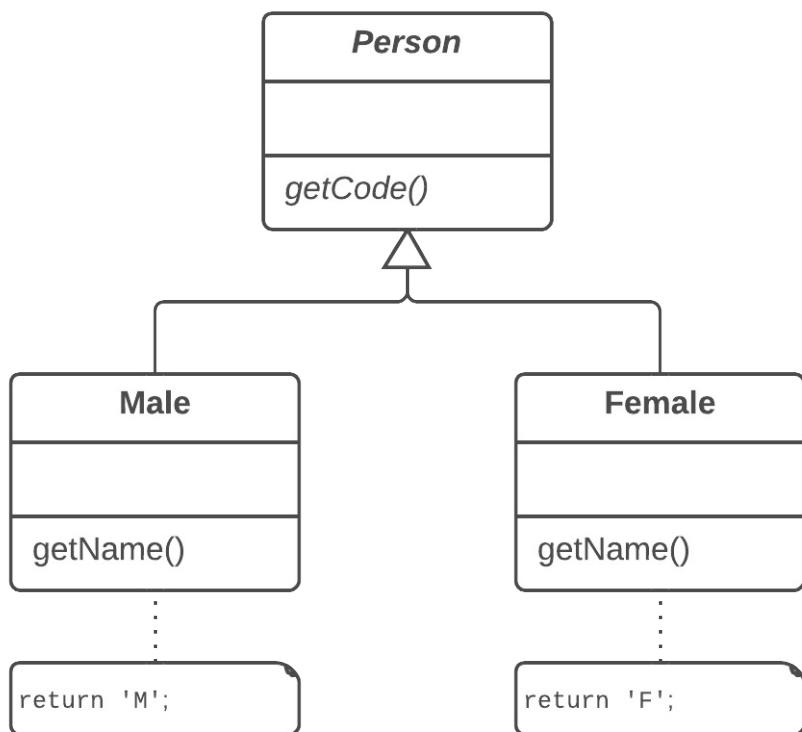


# ☒ Замена подкласса полями

Также известен как *Replace Subclass with Fields*

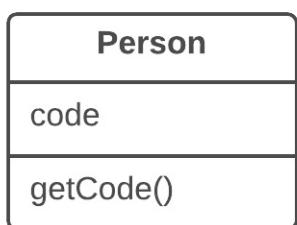
## Проблема

У вас есть подклассы, которые отличаются только методами, возвращающими данные-константы.



## Решение

Замените методы полями в родительском классе и удалите подклассы.



## Причины рефакторинга

Бывает так, что вам нужно развернуть действие рефакторинга избавления от кодирования типа.

В одном из подобных случаев иерархия подклассов может отличаться только значениями, возвращаемыми определёнными методами. Причём эти значения не являются результатом вычисления, а жёстко прописаны либо в самих методах, либо в полях, возвращаемых методами. Чтобы упростить архитектуру классов, такая иерархия может быть свёрнута в один класс, содержащий одно или несколько полей с нужными значениями в зависимости от ситуации.

Нужда в этих изменениях могла возникнуть после перемещения большого числа функциональностей из иерархии классов куда-то в другое место. После этого текущая иерархия потеряла свою ценность, и подклассы стали создавать только избыточную сложность.

## Достоинства

- Упрощает архитектуру системы. Создание подклассов – слишком избыточное решение, если все, что нужно сделать, это возвращать разные значения в нескольких методах.

## Порядок рефакторинга

1. Примените к подклассам замену конструктора фабричным методом.
2. Замените вызовы конструкторов подклассов вызовами фабричного метода суперкласса.
3. Объявите в суперклассе поля для хранения значений каждого из методов подклассов, возвращающих константные значения.
4. Создайте защищённый конструктор суперкласса для инициализации новых полей.
5. Создайте или модифицируйте имеющиеся конструкторы подклассов, чтобы они вызывали новый конструктор родительского класса и передавали в него соответствующие значения.
6. Реализуйте каждый константный метод в родительском классе так, чтобы он возвращал значение соответствующего поля, а затем удалите метод из подкласса.
7. Если конструктор подкласса имеет какую-то дополнительную функциональность, примените встраивание метода для встраивания его конструктора в фабричный метод суперкласса.
8. Удалите подкласс.

# Анти-рефакторинг

## § Замена кодирования типа подклассами





# Упрощение условных выражений

Логика условного выполнения имеет тенденцию становиться сложной, поэтому ряд рефакторингов направлен на то, чтобы упростить ее.

## § Разбиение условного оператора

**Проблема:** У вас есть сложный условный оператор (`if`-`then` / `else` ИЛИ `switch`).

**Решение:** Выделите в отдельные методы все сложные части оператора: условие, `then` И `else`.

## § Объединение условных операторов

**Проблема:** У вас есть несколько условных операторов, ведущих к одному результату или действию.

**Решение:** Объедините все условия в одном условном операторе.

## § Объединение дублирующихся фрагментов в условных операторах

**Проблема:** Однаковый фрагмент кода находится во всех ветках условного оператора.

**Решение:** Вынесите его за рамки оператора.

## § Удаление управляющего флага

**Проблема:** У вас есть булевская переменная, которая играет роль управляющего флага для нескольких булевых выражений.

**Решение:** Используйте `break`, `continue` И `return` вместо этой переменной.

## § Замена вложенных условных операторов граничным оператором

**Проблема:** У вас есть группа вложенных условных операторов, среди которых сложно выделить нормальный ход выполнения кода.

**Решение:** Выделите все проверки специальных или граничных случаев выполнения в отдельные условия и поместите их перед основными проверками. В идеале, вы должны получить «плоский» список условных операторов, идущих один за другим.

## § Замена условного оператора полиморфизмом

**Проблема:** У вас есть условный оператор, который, в зависимости от типа или свойств объекта, выполняет различные действия.

**Решение:** Создайте подклассы, которым соответствуют ветки условного оператора. В них создайте общий метод и переместите в него код из соответствующей ветки условного оператора. Впоследствии замените условный оператор на вызов этого метода. Таким образом, нужная реализация будет выбираться через полиморфизм в зависимости от класса объекта.

## § Введение Null-объекта

**Проблема:** Из-за того, что некоторые методы возвращают `null` вместо реальных объектов, у вас в коде присутствует множество проверок на `null`.

**Решение:** Вместо `null` возвращайте Null-объект, который предоставляет поведение по умолчанию.

## § Введение проверки утверждения

**Проблема:** Корректная работа участка кода предполагает наличие каких-то определённых условий или значений.

**Решение:** Замените эти предположения конкретными проверками.



# ❖ Разбиение условного оператора

Также известен как *Decompose Conditional*

## Проблема

У вас есть сложный условный оператор (`if-then / else` или `switch`).

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

## Решение

Выделите в отдельные методы все сложные части оператора: условие, `then` и `else`.

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

## Причины рефакторинга

Чем длиннее кусок кода, тем сложнее понять, что он делает. Все усложняется ещё больше, когда код щедро приправлен условными операторами:

- пока вы разберётесь в том, что делает код в `then`, вы забываете, какое условие стояло в операторе;
- пока вы разбираетесь с `else`, вы забываете, что делал код в `then`.

## Достоинства

- Извлекая код условного оператора в методы с понятным названием, вы упрощаете жизнь тому, кто впоследствии будет этот код поддерживать (зачастую вам самим через месяц или два).
- Кстати, этот рефакторинг применим и для коротких выражений в условиях оператора. Стока `isSalaryDay()` куда наглядней опишет то, что она делает, чем код сравнения дат.

## Порядок рефакторинга

1. Выделите условие в отдельный метод с помощью выделения метода.
2. Повторите выделение для `then` и `else` части оператора.

## Борется с запахом

### § Длинный метод



# ✖ Объединение условных операторов

Также известен как *Consolidate Conditional Expression*

## Проблема

У вас есть несколько условных операторов, ведущих к одинаковому результату или действию.

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

## Решение

Объедините все условия в одном условном операторе.

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

## Причины рефакторинга

Код содержит множество чередующихся операторов, которые выполняют одинаковые действия. Причина разделения операторов неочевидна.

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

## Достоинства

- Убирает дублирование управляющего кода. Объединение множества условных операторов, ведущих к одной цели, помогает показать, что на самом деле вы делаете только одну сложную проверку, ведущую к одному общему действию.
- Объединив все операторы в одном, вы позволяете выделить это сложное условие в новый метод с названием, отражающим суть этого выражения.

## Порядок рефакторинга

Перед тем как осуществлять рефакторинг, убедитесь, что в условиях операторов нет «побочных эффектов», или, другими словами, они не модифицируют что-то, а только возвращают значения. Побочные эффекты могут быть и в коде, который выполняется внутри самого оператора. Например, по результатам условия, что-то добавляется к переменной.

1. Объедините множество условий в одном с помощью операторов `и` и `или`.  
Объединение операторов обычно следует такому правилу:

- Вложенные условия соединяются с помощью оператора `и`.
  - Условия, следующие друг за другом, соединяются с помощью оператора `или`.
2. Извлеките метод из условия оператора и назовите его так, чтобы он отражал суть проверяемого выражения.

## Борется с запахом

### § Дублирование кода



# Объединение дублирующихся фрагментов в условных операторах

Также известен как *Consolidate Duplicate Conditional Fragments*

## Проблема

Одинаковый фрагмент кода находится во всех ветках условного оператора.

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

## Решение

Вынесите его за рамки оператора.

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

## Причины рефакторинга

Дублирующий код находится внутри всех веток условного оператора. Зачастую это является результатом эволюции кода внутри веток оператора, тем более, если над кодом работало несколько человек.

## Достоинства

- Убивает дублирование кода.

## Порядок рефакторинга

1. Если дублирующие участки находятся в начале веток оператора, вынесите их перед условным оператором.
2. Если такой код выполняется в конце веток, поместите его после условного оператора.
3. Если дублирующий код расположен случайным образом внутри веток, вам нужно для начала попытаться передвинуть его в начало или в конец ветки, в зависимости от того, меняет ли он результат последующего кода.
4. Дублирующий фрагмент кода более одной строки можно попытаться извлечь в новый метод, если в этом есть смысл.

## Борется с запахом

### § Дублирование кода



# ✖ Удаление управляющего флага

Также известен как *Remove Control Flag*

## Проблема

У вас есть булевская переменная, которая играет роль управляющего флага для нескольких булевых выражений.

## Решение

Используйте `break`, `continue` и `return` вместо этой переменной.

## Причины рефакторинга

Управляющие флаги пришли к нам из тех «бородатых» дней, когда хорошим стилем программирования считалось иметь в функции одну входную точку (строку объявления функции) и одну выходную точку (в самом конце функции).

В современных языках программирования этот подход устарел, т.к. у нас появились специальные операторы для управления ходом программы в циклах и других сложных конструкциях:

- `break` : останавливает выполнение цикла;
- `continue` : останавливает выполнение текущего витка цикла и переходит к проверке условия цикла и следующей итерации;
- `return` : останавливает выполнение всей функции и возвращает её результат, если он подан в этом операторе.

## Достоинства

- Код с управляющим флагом зачастую получается значительно более запутанным, чем при использовании операторов управления выполнением.

## Порядок рефакторинга

1. Найдите присваивание значения управляющему флагу, которое приводит к выходу из цикла или текущей итерации.
2. Замените его на `break`, если это выход из цикла, или `continue`, если это выход из итерации, или `return`, если нужно вернуть это значение из функции.
3. Уберите весь остальной код и проверки, связанные с управляющим флагом.



# ☒ Замена вложенных условных операторов граничным оператором

Также известен как *Replace Nested Conditional with Guard Clauses*

## Проблема

У вас есть группа вложенных условных операторов, среди которых сложно выделить нормальный ход выполнения кода.

```
public double getPayAmount() {  
    double result;  
  
    if (isDead) {  
  
        result = deadAmount();  
  
    }  
  
    else {  
  
        if (isSeparated) {  
  
            result = separatedAmount();  
  
        }  
  
        else {  
  
            if (isRetired){  
  
                result = retiredAmount();  
  
            }  
  
            else{  
  
                result = normalPayAmount();  
  
            }  
  
        }  
  
    }  
  
    return result;  
}
```

## Решение

Выделите все проверки специальных или граничных случаев выполнения в отдельные условия и поместите их перед основными проверками. В идеале, вы

должны получить «плоский» список условных операторов, идущих один за другим.

```
public double getPayAmount() {  
    if (isDead) {  
        return deadAmount();  
    }  
    if (isSeparated) {  
        return separatedAmount();  
    }  
    if (isRetired) {  
        return retiredAmount();  
    }  
    return normalPayAmount();  
}
```

## Причины рефакторинга

«Условный оператор из ада» довольно просто отличить. Отступы каждого из уровней вложенности формируют в нем отчётливую стрелку, указывающую вправо:

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

Разобраться в том, что и как делает такой оператор довольно сложно, так как «нормальный» ход выполнения в нем не очевиден. Такие операторы появляются эволюционным путём, когда каждое из условий добавляется в разные промежутки времени без мыслей об оптимизации остальных условий.

Чтобы упростить такой оператор, нужно выделить все особые случаи в отдельные условные операторы, которые бы при наступлении граничных условий, сразу заканчивали выполнение и возвращали нужное значение. По сути, ваша цель – сделать такой оператор плоским.

## Порядок рефакторинга

Постарайтесь избавиться от «побочных эффектов» в условиях операторов. Разделение запроса и модификатора может в этом помочь. Такое решение понадобится для дальнейших перестановок условий.

1. Выделите граничные условия, которые приводят к вызову исключения или немедленному возвращению значения из метода. Переместите эти условия в начало метода.
2. После того, как с переносами покончено, и все тесты стали проходить, проверьте, можно ли использовать **объединение условных операторов** для граничных условных операторов, ведущих к одинаковым исключениям или возвращаемым значениям.



# ☒ Замена условного оператора полиморфизмом

Также известен как *Replace Conditional with Polymorphism*

## Проблема

У вас есть условный оператор, который, в зависимости от типа или свойств объекта, выполняет различные действия.

```
class Bird {  
    //...  
  
    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");  
    }  
}
```

## Решение

Создайте подклассы, которым соответствуют ветки условного оператора. В них создайте общий метод и переместите в него код из соответствующей ветки условного оператора. Впоследствии замените условный оператор на вызов этого метода. Таким образом, нужная реализация будет выбираться через полиморфизм в зависимости от класса объекта.

```
abstract class Bird {  
    //...  
    abstract double getSpeed();  
}  
  
class European extends Bird {  
    double getSpeed() {  
        return getBaseSpeed();  
    }  
}  
  
class African extends Bird {  
    double getSpeed() {  
        return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;  
    }  
}  
  
class NorwegianBlue extends Bird {  
    double getSpeed() {  
        return (isNailed) ? 0 : getBaseSpeed(voltage);  
    }  
}  
  
// Somewhere in client code  
speed = bird.getSpeed();
```

## Причины рефакторинга

Этот рефакторинг может помочь, если у вас в коде есть условные операторы, которые выполняют различную работу, в зависимости от:

- класса объекта или интерфейса, который он реализует;
- значения какого-то из полей объекта;
- результата вызова одного из методов объекта.

При этом если у вас появится новый тип или свойство объекта, нужно будет искать и добавлять код во все схожие условные операторы. Таким образом,

польза от данного рефакторинга увеличивается, если условных операторов больше одного, и они разбросаны по всем методам объекта.

## Достоинства

- Этот рефакторинг реализует принцип *говори, а не спрашивай*: вместо того, чтобы спрашивать объект о его состоянии, а потом выполнять на основании этого какие-то действия, гораздо проще просто сказать ему, что нужно делать, а как это делать он решит сам.
- Убивает дублирование кода. Вы избавляетесь от множества почти одинаковых условных операторов.
- Если вам потребуется добавить новый вариант выполнения, все, что придётся сделать, это добавить новый подкласс, не трогая существующий код (*принцип открытости/закрытости*).

## Порядок рефакторинга

### Подготовка к рефакторингу

Чтобы выполнить этот рефакторинг, вам следует иметь готовую иерархию классов, в которых будут содержаться альтернативные поведения. Если такой иерархии ещё нет, нужно создать её. В этом могут помочь другие рефакторинги:

- **Замена кодирования типа подклассами.** При этом для всех значений какого-то свойства объекта будут созданы свои подклассы. Это хоть и простой, но менее гибкий способ, т.к. нельзя будет создать подклассы для других свойств объекта.
- **Замена кодирования типа состоянием/стратегией.** При этом для определенного свойства объекта будет выделен свой класс и из него созданы подклассы для каждого значения этого свойства. Текущий класс будет содержать ссылки на объекты такого типа и делегировать им выполнение.

Последующие шаги этого рефакторинга подразумевают, что вы уже создали иерархию.

### Шаги рефакторинга

1. Если условный оператор находится в методе, который выполняет ещё какие-то действия, извлеките его в новый метод.
2. Для каждого подкласса иерархии, переопределите метод, содержащий условный оператор, и скопируйте туда код соответствующей ветки оператора.
3. Удалите эту ветку из условного оператора.

4. Повторяйте замену, пока условный оператор не опустеет. Затем удалите условный оператор и объявите метод абстрактным.

## Борется с запахом

### § Операторы switch





# ✖ Введение Null-объекта

Также известен как *Introduce Null Object*

## Проблема

Из-за того, что некоторые методы возвращают `null` вместо реальных объектов, у вас в коде присутствует множество проверок на `null`.

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

## Решение

Вместо `null` возвращайте Null-объект, который предоставляет поведение по умолчанию.

```

class NullCustomer extends Customer {

    booleanisNull() {
        return true;
    }

    Plan getPlan() {
        return new NullPlan();
    }

    // Some other NULL functionality.
}

// Replace null values with Null-object.
customer = (order.customer != null) ?
    order.customer : new NullCustomer();

// Use Null-object as if it's normal subclass.
plan = customer.getPlan();

```

## Причины рефакторинга

Десятки проверок на `null` усложняют и засоряют код.

## Недостатки

- За отказ от условных операторов вы платите ещё одним новым классом.

## Порядок рефакторинга

1. Из интересующего вас класса создайте подкласс, который будет выполнять роль Null-объекта.
2. В обоих классах создайте метод `isNull()`, который будет возвращать `true` для Null-объекта и `false` для реального класса.
3. Найдите все места, где код может вернуть `null` вместо реального объекта. Измените этот код так, чтобы он возвращал Null-объект.
4. Найдите все места, где переменные реального класса сравниваются с `null`. Замените такие проверки вызовом метода `isNull()`.
- 5.

- Если в этих условных операторах при значении не равном `null` выполняются методы исходного класса, переопределите эти методы в Null-классе и вставьте туда код из `else` части условия. После этого условный оператор можно будет вообще удалить, а разное поведение будет осуществляться за счёт полиморфизма.
- Если не все так просто, и методы переопределить не получается, посмотрите, можно ли просто выделить операции, которые должны были выполняться при значении равном `null` в новые методы Null-объекта. Вызывайте эти методы вместо старого кода в `else` как операции по умолчанию.

## Родственные рефакторинги

### § Замена условного оператора полиморфизмом

## Реализует паттерн проектирования

### § Null-object

## Борется с запахом

### § Операторы switch



### § Временное поле





# ✖ Введение проверки утверждения

Также известен как *Introduce Assertion*

Здесь под проверками подразумевается использование вызовов `assert()`.

## Проблема

Корректная работа участка кода предполагает наличие каких-то определённых условий или значений.

```
double getExpenseLimit() {  
    // should have either expense limit or a primary project  
    return (expenseLimit != NULL_EXPENSE) ?  
        expenseLimit:  
        primaryProject.getMemberExpenseLimit();  
}
```

## Решение

Замените эти предположения конкретными проверками.

```
double getExpenseLimit() {  
    Assert.isTrue(expenseLimit != NULL_EXPENSE || primaryProject != null);  
  
    return (expenseLimit != NULL_EXPENSE) ?  
        expenseLimit:  
        primaryProject.getMemberExpenseLimit();  
}
```

## Причины рефакторинга

Участок кода создает предположения о чем-то (например, о текущем состоянии объекта, значении параметра или локальной переменной). Обычно это предположение никогда не будет нарушено разве что при наличии какой-то ошибки.

Сделайте эти предположения явными, добавив соответствующие проверки. Как и подсказки типов в параметрах методов, эти проверки могут играть роль живой документации кода.

Хорошим признаком того, что нужна проверка каких-то условий, являются комментарии в коде. Если вы видите комментарии, описывающие условия, при которых метод корректно работать, значит здесь неплохо было бы вставить проверку-утверждение этих условий.

## Достоинства

- Если какое-то предположение неверно и в результате код производит также неверный результат, лучше сразу остановить выполнение, пока это не привело к фатальным последствиям и порче данных. Кроме того, это означает, что вы упустили написание какого-то теста в наборе тестов вашей программы.

## Недостатки

- Иногда вместо простой проверки лучше вставить исключение. При этом вы можете выбрать необходимый класс исключения и дать остальному коду возможность корректно его обработать.
- В каких случаях исключение лучше простой проверки? Если к нему может привести деятельность пользователя или системы, и вы можете обработать это исключение. С другой стороны, обычные безымянные необрабатываемые исключения равносильны простым проверкам – вы их не обрабатываете, они могут быть вызваны только как результат бага в программе, который никогда не должен был возникнуть.

## Порядок рефакторинга

Когда вы видите, что предполагается какое-то условие, добавьте проверку этого условия, чтобы проверить его наверняка.

Добавление проверки не должно изменять поведение программы.

Не перебарщивайте с использованием проверок для **всего**, что только можно, на выбранном участке кода. Нужно проверять лишь те условия, без которых код перестанет корректно работать. Если ваш код нормально работает, в том числе, в случае, когда проверка не проходит успешно, такую проверку можно убрать.

## Борется с запахом



# Упрощение вызовов методов

Эти рефакторинги делают вызовы методов проще и яснее для понимания. Это, в свою очередь, упрощает интерфейсы взаимодействия между классами.

## § Переименование метода

**Проблема:** Название метода не раскрывает суть того, что он делает.

**Решение:** Измените название метода.

## § Добавление параметра

**Проблема:** Методу не хватает данных для осуществления каких-то действий.

**Решение:** Создайте новый параметр, чтобы передать эти данные.

## § Удаление параметра

**Проблема:** Параметр не используется в теле метода.

**Решение:** Удалите неиспользуемый параметр.

## § Разделение запроса и модификатора

**Проблема:** У вас есть метод, который возвращает какое-то значение, но при этом в процессе работы он изменяет что-то внутри объекта.

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

## § Параметризация метода

**Проблема:** Несколько методов выполняют похожие действия, которые отличаются только какими-то внутренними значениями, числами или операциями.

**Решение:** Объедините все эти методы в один с параметром, в который будет передаваться отличающееся значение.

## § Замена параметра набором специализированных методов

**Проблема:** Метод разбит на части, каждая из которых выполняется в зависимости от значения какого-то параметра.

**Решение:** Извлеките отдельные части метода в собственные методы и вызывайте их вместо оригинального метода.

## § Передача всего объекта

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

**Решение:** Вместо этого передавайте весь объект.

## § Замена параметра вызовом метода

**Проблема:** Вызываем метод и передаем его результаты как параметры другого метода. При этом значение параметров могли бы быть получены и внутри вызываемого метода.

**Решение:** Вместо передачи значения через параметры метода, попробуйте переместить код получения значения внутрь самого метода.

## § Замена параметров объектом

**Проблема:** В ваших методах встречается повторяющаяся группа параметров.

**Решение:** Замените эти параметры объектом.

## § Удаление сеттера

**Проблема:** Значение поля должно быть установлено только в момент создания и больше никогда не меняться.

**Решение:** Удалите методы, устанавливающие значение этого поля.

## § Скрытие метода

**Проблема:** Метод не используется другими классами либо используется только внутри своей иерархии классов.

**Решение:** Сделайте метод приватным или защищенным.

## § Замена конструктора фабричным методом

**Проблема:** У вас есть сложный конструктор, делающий нечто большее, чем простая установка значений полей объекта.

**Решение:** Создайте фабричный метод и замените им вызовы конструктора.

## § Замена кода ошибки исключением

**Проблема:** Метод возвращает определенное значение, которое будет сигнализировать об ошибке.

**Решение:** Вместо этого следует выбрасывать исключение.

## § Замена исключения проверкой условия

**Проблема:** Вы выбрасываете исключение там, где можно было бы обойтись простой проверкой условия.

**Решение:** Замените выбрасывание исключения проверкой этого условия.

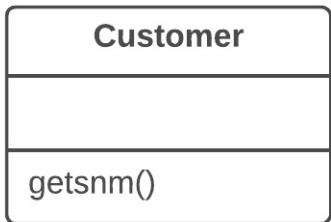


# ✖ Переименование метода

Также известен как *Rename Method*

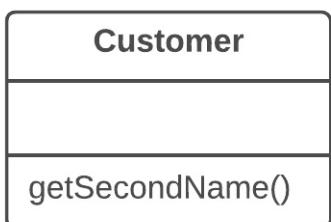
## Проблема

Название метода не раскрывает суть того, что он делает.



## Решение

Измените название метода.



## Причины рефакторинга

Метод мог получить неудачное название с самого начала. Например, кто-то создал метод вспыхах, не придал должного значения хорошему названию.

С другой стороны, метод мог быть назван удачно изначально, но ввиду расширения его функциональности, имя метода перестало быть актуальным.

## Достоинства

- Улучшает читабельность кода. Постарайтесь дать новому методу такое название, которое бы отражало суть того, что он делает. Например,  
`createOrder()`, `renderCustomerInfo()` и т.д.

# Порядок рефакторинга

1. Проверьте, не определён ли метод в суперклассе или подклассе. Если так, нужно будет повторить все шаги и в этих классах.
2. Следующий шаг важен, чтобы сохранить работоспособность программы во время рефакторинга. Итак, 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. Этот шаг неосуществим, если старый метод является частью публичного интерфейса. В этом случае, старый метод нужно пометить как устаревший (`deprecated`).

# Родственные рефакторинги

§ [Добавление параметра](#)

§ [Удаление параметра](#)

# Борется с запахом

§ [Альтернативные классы с разными интерфейсами](#)

§ [Комментарии](#)

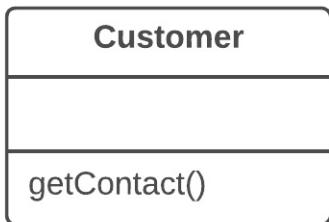


# ⌘ Добавление параметра

Также известен как Add Parameter

## Проблема

Методу не хватает данных для осуществления каких-то действий.



## Решение

Создайте новый параметр, чтобы передать эти данные.



## Причины рефакторинга

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

## Достоинства

- Введение нового параметра всегда соперничает с введением нового приватного поля, которое бы содержало необходимые методу данные. Исходя из этого, можно сказать, что поле лучше добавить тогда, когда вам понадобятся какие-то эпизодические или часто изменяющиеся данные, которые нет смысла держать в объекте все время. В этом случае новый параметр послужит лучше приватного поля и рефакторинг будет оправданным. В других случаях лучше

ввести приватное поле и заполнять его нужными данными перед вызовом метода.

## Недостатки

- Добавить новый параметр всегда легче, чем его убрать, поэтому списки параметров часто разрастаются до неприличных размеров. Это приводит к появлению запаха длинный список параметров.
- То, что вам потребовалось добавить новый параметр, иногда означает, что ваш класс не содержит необходимых данных либо существующие параметры не несут необходимых связанных данных. В обоих случаях, лучшим решением было бы подумать о перемещении данных в основной класс либо в другие классы, объекты которых уже доступны внутри метода.

## Порядок рефакторинга

1. Проверьте, не определён ли метод в суперклассе или подклассе. Если метод в них присутствует, нужно будет повторить все шаги также в этих классах.
2. Следующий шаг важен, чтобы сохранить работоспособность программы во время рефакторинга. Итак, создайте новый метод, скопировав старый, и добавьте в него требуемый параметр. Замените код старого метода вызовом нового метода. Вы можете подставить любое значение в новый параметр (например `null` для объектов или ноль для чисел).
3. Найдите все обращения к старому методу и замените их обращениями к новому методу.
4. Удалите старый метод. Этот шаг неосуществим, если старый метод является частью публичного интерфейса. В этом случае старый метод нужно пометить как устаревший (`deprecated`).

## Анти-рефакторинг

### § Удаление параметра

## Родственные рефакторинги

### § Переименование метода

## Помогает рефакторингу

### § Замена параметров объектом



# ✖ Удаление параметра

Также известен как Remove Parameter

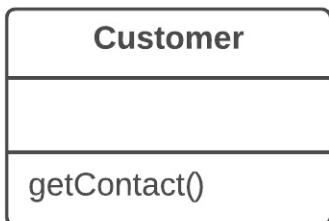
## Проблема

Параметр не используется в теле метода.



## Решение

Удалите неиспользуемый параметр.



## Причины рефакторинга

Каждый параметр в вызове метода заставляет человека, пишущего код метода, размышлять о том, какая информация может оказаться в этом параметре. И если параметр потом вовсе не используется в теле метода, значит, весь этот мыслительный процесс уходит впустую.

Кроме того, дополнительные параметры – это ещё и лишний код к выполнению.

Порой мы добавляем параметры на будущее, предчувствуя скорые изменения в методе, для которых потребуется этот параметр. Тем не менее, опыт показывает, что лучше добавить параметр тогда, когда он действительно понадобится, ведь ожидаемые изменения могут так и не наступить.

## Достоинства

- Метод должен содержать только действительно необходимые параметры.

## Когда нельзя применить

- Не стоит начинать этот рефакторинг, если метод имеет альтернативные реализации в подклассах или в суперклассе, и ваш параметр используется в этих реализациях.

## Порядок рефакторинга

1. Проверьте, не определён ли метод в суперклассе или подклассе. Если так, используется ли там параметр? Если в какой-то из реализаций параметр используется, воздержитесь от рефакторинга.
2. Следующий шаг важен, чтобы сохранить работоспособность программы во время рефакторинга. Итак, создайте новый метод, скопировав старый, и удалите из него требуемый параметр. Замените код старого метода вызовом нового метода.
3. Найдите все обращения к старому методу и замените их обращениями к новому методу.
4. Удалите старый метод. Этот шаг неосуществим, если старый метод является частью публичного интерфейса. В этом случае, старый метод нужно пометить как устаревший (`deprecated`).

## Анти-рефакторинг

### § Добавление параметра



## Родственные рефакторинги

### § Переименование метода



## Помогает рефакторингу

### § Замена параметра вызовом метода



## Борется с запахом

### § Теоретическая общность



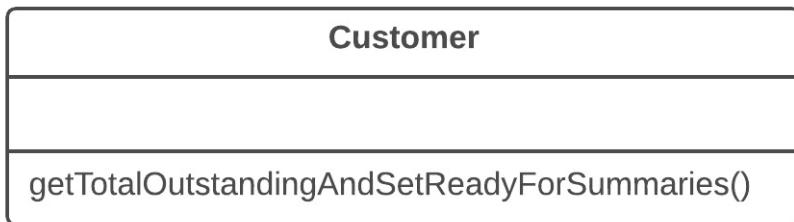


# ✖ Разделение запроса и модификатора

Также известен как *Separate Query from Modifier*

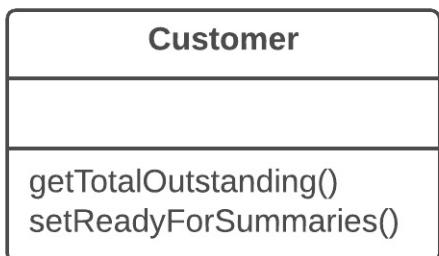
## Проблема

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



## Решение

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



## Причины рефакторинга

Этот рефакторинг является реализацией *принципа разделения команд от запросов данных*. Суть принципа сводится к тому, чтобы отделять 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**. Когда *запрос* и *модификатор* совмещены, у вас нет способа получить данные из объекта без того, чтобы не внести изменения в его состояние. Другими словами, вы задаёте вопрос и можете изменить ответ прямо в процессе его получения. Усугубляется эта проблема тем,

что человек, вызывающий запрос, может не знать о «побочных действиях» такого метода, что нередко приводит к ошибкам выполнения программы.

Однако стоит подчеркнуть, что «побочными действиями» опасны только те *модификаторы*, которые меняют **видимое** состояние объекта. Это, например, поля, которые доступны из публичного интерфейса объекта, записи в базе данных, в файлах и т.д. Если какой-то *модификатор* всего лишь кеширует какую-то сложную операцию и сохраняет её внутри приватного поля класса, он вряд ли приведёт к «побочным действиям».

## Достоинства

- Если у вас есть запрос, который не меняет состояние программы, вы можете вызывать его сколько угодно раз, не опасаясь того, что результат измениться 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.

## Порядок рефакторинга

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*, но сразу перед этой строкой вставьте вызов оригинального метода.
4. Избавьтесь от кода возврата значения в оригинальном методе. После этого он станет правильным *методом-модификатором*.

## Помогает рефакторингу

### § Замена переменной вызовом метода

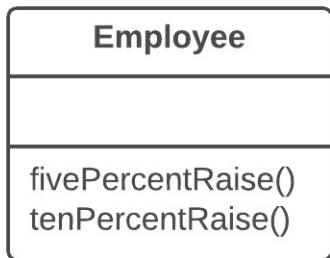


# ❖ Параметризация метода

Также известен как *Parameterize Method*

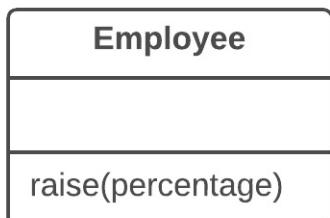
## Проблема

Несколько методов выполняют похожие действия, которые отличаются только какими-то внутренними значениями, числами или операциями.



## Решение

Объедините все эти методы в один с параметром, в который будет передаваться отличающееся значение.



## Причины рефакторинга

Если у вас есть схожие методы, скорей всего, в них присутствует дублирующий код со всеми вытекающими недостатками.

Кроме того, если вам нужно будет добавить ещё одну вариацию функциональности, вам придётся создавать ещё один метод. Вместо этого можно было запустить существующий метод с другим параметром.

## Недостатки

- Иногда при проведении рефакторинга можно переусердствовать, в результате чего у вас появится длинный и сложный общий метод вместо нескольких простых.
- Кроме того, будьте осторожны, выделяя в параметр переключатель какой-то функциональности. В дальнейшем это может привести к созданию большого условного оператора, который надо будет лечить с помощью замены параметра набором специализированных методов.

## Порядок рефакторинга

1. Создайте новый метод с параметром и поместите в него общий для всех методов код, применяя извлечение метода. Обратите внимание, иногда общей оказывается только определённая часть методов. В этом случае рефакторинг сведётся к извлечению только этой общей части в новый метод.
2. Отличающееся значение замените параметром в коде нового метода.
3. Для каждого старого метода найдите места, где они вызываются, и поменяйте их вызовы на вызовы нового метода с параметром. После чего старый метод можно удалить.

## Анти-рефакторинг

### § Замена параметра набором специализированных методов

## Родственные рефакторинги

### § Извлечение метода

### § Создание шаблонного метода

## Борется с запахом

### § Дублирование кода



# ☒ Замена параметра набором специализированных методов

Также известен как *Replace Parameter with Explicit Methods*

## Проблема

Метод разбит на части, каждая из которых выполняется в зависимости от значения какого-то параметра.

```
void setValue(String name, int value) {  
    if (name.equals("height")) {  
        height = value;  
        return;  
    }  
    if (name.equals("width")) {  
        width = value;  
        return;  
    }  
    Assert.shouldNeverReachHere();  
}
```

## Решение

Извлеките отдельные части метода в собственные методы и вызывайте их вместо оригинального метода.

```
void setHeight(int arg) {  
    height = arg;  
}  
  
void setWidth(int arg) {  
    width = arg;  
}
```

## Причины рефакторинга

Метод, содержащий варианты выполнения, разросся до грандиозных размеров. В каждой из таких цепочек выполняется нетривиальный код. При этом новые варианты добавляются очень редко.

## Достоинства

- Улучшает читабельность кода. Куда очевидней, что делает метод

```
startEngine()  чем setValue("engineEnabled", true) .
```

## Когда нельзя применить

- Не стоит применять замену параметра явными методами, если метод редко меняется, а новые вариации внутри него не добавляются.

## Порядок рефакторинга

1. Для каждого варианта исполнения метода создайте свой метод. Запускайте эти методы в зависимости от значения параметра в основном методе.
2. Найдите все места, где вызывается оригиналный метод. Подставьте туда вызов одного из новых методов в зависимости от передающегося параметра.
3. Когда не останется ни одного вызова оригинального метода, его можно будет удалить.

## Анти-рефакторинг

### § Параметризация метода



## Родственные рефакторинги

### § Замена условного оператора полиморфизмом

## Борется с запахом

### § Операторы switch



### § Длинный метод





# Передача всего объекта

Также известен как *Preserve Whole Object*

## Проблема

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

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

## Решение

Вместо этого передавайте весь объект.

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

## Причины рефакторинга

Проблема заключается в том, что перед вызовом вашего метода нужно каждый раз вызывать методы будущего объекта-параметра. Если способ вызова этих методов либо количество данных, получаемых для метода, поменяется, то изменения придётся искать и вносить в дюжину мест по всей программе.

Вместо этого код получения всех нужных данных может храниться одном месте – в самом методе.

## Достоинства

- Вместо пачки разнообразных параметров вы видите 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

- В некоторых случаях после такого преобразования метод теряет в универсальности, т.к. он мог получать данные из множества разных

источников, а в результате рефакторинга мы ограничиваем круг его применения только для объектов с определённым интерфейсом.

## Порядок рефакторинга

1. Создайте параметр в методе для объекта, из которого можно получить нужные значения.
2. Теперь начинайте по одному удалять 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.

## Родственные рефакторинги

- § [Замена параметров объектом](#)
- § [Замена параметра вызовом метода](#)

## Борется с запахом

- § [Одержанность элементарными типами](#)
- § [Длинный список параметров](#)
- § [Длинный метод](#)
- § [Группы данных](#)



# ☒ Замена параметра вызовом метода

Также известен как *Replace Parameter with Method Call*

## Проблема

Вызываем метод и передаем его результаты как параметры другого метода. При этом значение параметров могли бы быть получены и внутри вызываемого метода.

```
int basePrice = quantity * itemPrice;  
  
double seasonDiscount = this.getSeasonalDiscount();  
  
double fees = this.getFees();  
  
double finalPrice = discountedPrice(basePrice, seasonDiscount, fees);
```

## Решение

Вместо передачи значения через параметры метода, попробуйте переместить код получения значения внутрь самого метода.

```
int basePrice = quantity * itemPrice;  
  
double finalPrice = discountedPrice(basePrice);
```

## Причины рефакторинга

В длинном списке параметров зачастую крайне сложно ориентироваться. Кроме того, вызовы таких методов часто превращаются в целую вереницу вычислений значений, которые будут передаваться в метод. Вот почему если значения параметра может быть вычислено при помощи вызова какого-то метода, это следует сделать внутри самого метода, а от параметра избавиться.

## Достоинства

- Избавляемся от лишних параметров, упрощая вызовы методов. Эти параметры зачастую создаются как задел на будущее (которое может так и не наступить).

## Недостатки

- Параметр может понадобиться завтра для каких-то других целей и метод придётся переписать.

## Порядок рефакторинга

1. Убедитесь, что код получения значения не использует параметров из текущего метода. Это важно, т.к. параметры текущего метода будут недоступны внутри другого метода, из-за чего перенос станет невозможен.
2. Если код получения значения сложнее, чем один вызов какого-то метода или функции, примените **извлечение метода**, чтобы выделить этот код в новый метод и сделать вызов простым.
3. В коде главного метода замените все обращения к заменяемому параметру вызовами метода получения значения.
4. Используйте **удаление параметра**, чтобы удалить неиспользуемый теперь параметр.

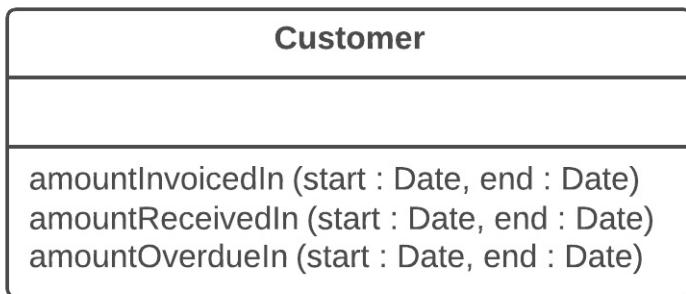


# Замена параметров объектом

Также известен как *Introduce Parameter Object*

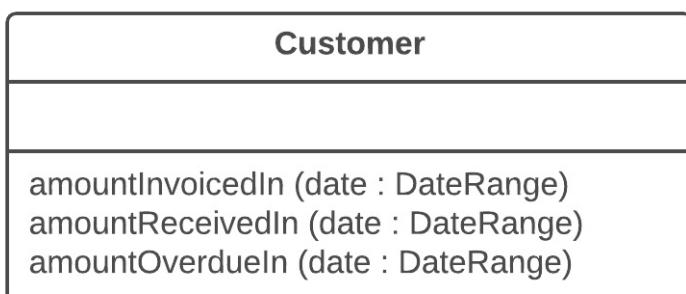
## Проблема

В ваших методах встречается повторяющаяся группа параметров.



## Решение

Замените эти параметры объектом.



## Причины рефакторинга

Однаковые группы параметров зачастую встречаются не в единственном методе. Это приводит к дублированию кода, как самих параметров, так и частых операций над ними. Как только вы сведёте параметры в одном классе, вы сможете переместить туда и методы обработки этих данных, очистив от этого кода другие методы.

## Достоинства

- Улучшает читабельность кода. Вместо пачки разнообразных параметров вы видите один объект с понятным названием.
- Одинаковые группы параметров тут и там создают особый род дублирования кода, при котором вроде бы не происходит вызова идентичного кода, но все время встречаются одинаковые группы параметров и аргументов.

## Недостатки

- Если вы переместили в новый класс только данные и не планируете перемещать в него никакие поведения и операции над этими данными, это может попахивать **классами данных**.

## Порядок рефакторинга

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 окончанию оцените, есть ли возможность и смысл перенести какую-то часть метода (а иногда и весь метод) в класс объекта-параметра. Если так, используйте **перемещение метода** или **извлечение метода**, чтобы осуществить перенос.

## Родственные рефакторинги

### § Передача всего объекта

## Борется с запахом

### § Длинный список параметров

### § Группы данных

### § Одержанность элементарными типами

### § Длинный метод

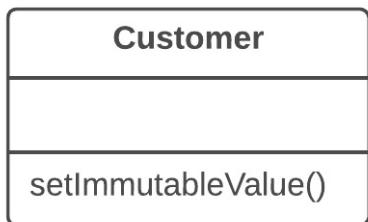


# ✂ Удаление сеттера

Также известен как *Remove Setting Method*

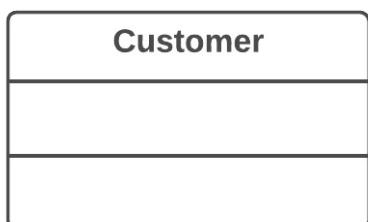
## Проблема

Значение поля должно быть установлено только в момент создания и больше никогда не меняться.



## Решение

Удалите методы, устанавливающие значение этого поля.



## Причины рефакторинга

Вы хотите сделать значение поля неизменяемым.

## Порядок рефакторинга

1. Значение поля должно меняться только в конструкторе. Если конструктор не содержит параметра для установки значения, нужно его добавить.
2. Найдите все вызовы сеттера.

- Если вызов сеттера стоит сразу после вызова конструктора текущего класса, переместите его аргумент в вызов конструктора и удалите сеттер.
- Вызовы сеттера в конструкторе замените на прямой доступ к полю.

3. Удалите сеттер.

## Помогает рефакторингу

### § Замена ссылки значением

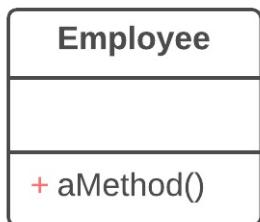


# ✂ Сокрытие метода

Также известен как *Hide Method*

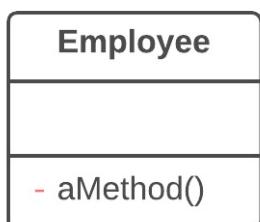
## Проблема

Метод не используется другими классами либо используется только внутри своей иерархии классов.



## Решение

Сделайте метод приватным или защищённым.



## Причины рефакторинга

Очень часто потребность в сокрытии методов получения и установки значений возникает в связи с разработкой более богатого интерфейса, предоставляющего дополнительное поведение. Особенно это проявляется в случае, когда вы начинали с класса в котором не было никаких методов, кроме геттеров и сеттеров.

По мере встраивания в класс нового поведения может обнаружиться, что в открытых геттерах и сеттерах более нет надобности, и тогда можно их скрыть. А после этого, если использовать только прямой доступ к полю, его методы доступа можно вообще удалить.

## Достоинства

- Сокрытие методов упрощает эволюционные изменения в вашем коде. Изменяя приватный метод, вам нужно будет заботиться только о том, чтобы не сломать текущий класс, т.к. этот метод не может быть использован где-то ещё.
- Делая методы приватными, вы подчёркиваете важность публичного интерфейса класса, другими словами, тех методов, которые остались публичными.

## Порядок рефакторинга

1. Регулярно делайте попытки найти методы, которые можно сделать приватными. Статический анализ кода и хорошее покрытие unit-тестами может очень помочь в этом.
2. Делайте каждый метод настолько приватным, насколько это возможно.

## Борется с запахом

§ Класс данных





# ☒ Замена конструктора фабричным методом

Также известен как *Replace Constructor with Factory Method*

## Проблема

У вас есть сложный конструктор, делающий нечто большее, чем простая установка значений полей объекта.

```
class Employee {  
    Employee(int type) {  
        this.type = type;  
    }  
    //...  
}
```

## Решение

Создайте фабричный метод и замените им вызовы конструктора.

```
class Employee {  
    static Employee create(int type) {  
        employee = new Employee(type);  
        // do some heavy lifting.  
        return employee;  
    }  
    //...  
}
```

## Причины рефакторинга

Самая очевидная причина применения этого рефакторинга связана с заменой кодирования типа подклассами.

У вас есть код, в котором раньше создавался объект, куда передавалось значение кодированного типа. После применения рефакторинга появилось уже несколько подклассов, из которых нужно создавать объекты в зависимости от значения кодированного типа. Изменить оригинальный конструктор так, чтобы он возвращал объекты подклассов, невозможно, поэтому мы создаём статический

фабричный метод, который будет возвращать объекты нужных классов, после чего он заменяет собой все вызовы оригинального конструктора.

Фабричные методы можно использовать и в других ситуациях, когда возможностей конструкторов оказывается недостаточно. Они важны при замене значения ссылкой. Их можно также применять для задания различных режимов создания, выходящих за рамки числа и типов параметров.

## Достоинства

- Фабричный метод не обязательно возвращает объект того класса, в котором он был вызван. Зачастую это могут быть его подклассы, выбираемые в зависимости от подаваемых в метод аргументов.
- Фабричный метод может иметь более удачное имя, описывающее, что и каким образом он возвращает, например, `Troops::GetCrew(myTank)`.
- Фабричный метод может вернуть уже созданный объект в отличие от конструктора, который всегда создает новый экземпляр.

## Порядок рефакторинга

1. Создайте фабричный метод. Поместите в его тело вызова текущего конструктора.
2. Замените все вызовы конструктора вызовами фабричного метода.
3. Объявите конструктор приватным.
4. Обследуйте код конструктора и попытайтесь вынести в фабричный метод тот код, который не относится к непосредственному конструированию объекта текущего класса.

## Помогает рефакторингу

- § Замена значения ссылкой
- § Замена кодирования типа подклассами

## Реализует паттерн проектирования

- § Фабричный метод





# ☒ Замена кода ошибки исключением

Также известен как *Replace Error Code with Exception*

## Проблема

Метод возвращает определенное значение, которое будет сигнализировать об ошибке.

```
int withdraw(int amount) {  
    if (amount > _balance) {  
        return -1;  
    }  
    else {  
        balance -= amount;  
        return 0;  
    }  
}
```

## Решение

Вместе этого следует выбрасывать исключение.

```
void withdraw(int amount) throws BalanceException {  
    if (amount > _balance) {  
        throw new BalanceException();  
    }  
    balance -= amount;  
}
```

## Причины рефакторинга

Возвращение кодов ошибок – давно устаревшая практика процедурного программирования. В современном программировании для обработки ошибок используются специальные классы, называемые исключениями. При возникновении проблемы вы «выбрасываете» такое исключение и оно впоследствии «ловится» одним из обработчиков исключений. При этом

запускается специальный код обработки внештатной ситуации, который игнорируется в обычных условиях.

## Достоинства

- Избавляет код от множества условных операторов проверки кодов ошибок. Обработчики исключений намного чётче разграничивают нормальный и нештатный путь исполнения программы.
- Классы исключений могут реализовывать собственные методы, а значит содержать часть функциональности по обработке ошибок (например, для перевода сообщений об ошибках).
- В отличие от исключений, коды ошибок не могут быть использованы в конструкторе, т.к. он должен возвращать только новый объект.

## Недостатки

- Обработку исключений можно превратить в `goto`-подобный костыль. Не делайте так! Не используйте исключения для управления исполнением кода. Исключения следует выбрасывать только в целях сообщения об ошибке или критической ситуации.

## Порядок рефакторинга

Старайтесь выполнять шаги этого рефакторинга только для одного кода ошибки за один раз. Так будет легче удержать в голове все важные сведения и избежать ошибок.

1. Найдите все вызовы метода, возвращающего код ошибки, и оберните его в `try / catch` блоки вместо проверки кода ошибки.
2. Внутри метода вместо возвращения кода ошибки выбрасывайте исключение.
3. Измените сигнатуру метода так, чтобы она содержала информацию о выбрасываемом исключении (секция `@throws`).



# ☒ Замена исключения проверкой условия

Также известен как *Replace Exception with Test*

## Проблема

Вы выбрасываете исключение там, где можно было бы обойтись простой проверкой условия.

```
double getValueForPeriod(int periodNumber) {  
    try {  
        return values[periodNumber];  
    } catch (ArrayIndexOutOfBoundsException e) {  
        return 0;  
    }  
}
```

## Решение

Замените выбрасывание исключения проверкой этого условия.

```
double getValueForPeriod(int periodNumber) {  
    if (periodNumber >= values.length) {  
        return 0;  
    }  
    return values[periodNumber];  
}
```

## Причины рефакторинга

Исключения должны использоваться для обработки внештатного поведения, связанного с неожиданной ошибкой. Они не должны служить заменой проверкам выполнения условий. Если исключения можно избежать, просто проверив какое-то условие перед выполнением действия, то стоит так и сделать. Исключения следует приберечь для настоящих ошибок.

Например, вы зашли на минное поле и там подорвались, вызвав исключение; исключение успешно обработалось и вас вынесло за пределы минного поля. Вместо этого можно бы было просто прочитать указатель перед минным полем и обойти его другой дорогой.

## Достоинства

- Простой условный оператор иногда может быть очевиднее блока обработки исключения.

## Порядок рефакторинга

1. Создайте условный оператор для граничного случая и поместите его перед `try / catch` блоком.
2. Переместите код из `catch`-секции внутрь этого условного оператора.
3. В `catch`-секции поставьте код выбрасывания обычного безымянного исключения и запустите все тесты.
4. Если никаких исключений не было выброшено во время тестов, избавьтесь от оператора `try / catch`.

## Родственные рефакторинги

### § Замена кода ошибки исключением



# Решение задач обобщения

Обобщение порождает собственную группу рефакторингов, в основном связанных с перемещением функциональности по иерархии наследования классов, создания новых классов и интерфейсов, а также замены наследования делегированием и наоборот.

## § Подъём поля

**Проблема:** Два класса имеют одно и то же поле.

**Решение:** Переместите поле в суперкласс, убрав его из подклассов.

## § Подъём метода

**Проблема:** Подклассы имеют методы, которые делают схожую работу.

**Решение:** В этом случае нужно сделать методы идентичными, а затем переместить их в суперкласс.

## § Подъём тела конструктора

**Проблема:** Подклассы имеют конструкторы с преимущественно одинаковым кодом.

**Решение:** Создайте конструктор в суперклассе и вынесите в него общий для подклассов код. Вызывайте конструктор суперкласса в конструкторах подкласса.

## § Спуск метода

**Проблема:** Поведение, реализованное в суперклассе, используется только одним или несколькими подклассами.

**Решение:** Переместите это поведение в подклассы.

## § Спуск поля

**Проблема:** Поле используется только в некоторых подклассах.

**Решение:** Переместите поле в эти подклассы.

## § Извлечение подкласса

**Проблема:** Класс имеет фичи, которые используются только в определённых случаях.

**Решение:** Создайте подкласс и используйте его в этих случаях.

## § Извлечение суперкласса

**Проблема:** У вас есть два класса с общими полями и методами.

**Решение:** Создайте для них общий суперкласс и перенесите туда одинаковые поля и методы.

## § Извлечение интерфейса

**Проблема:** Несколько клиентов пользуются одной и той же частью интерфейса класса. Либо в двух классах часть интерфейса оказалась общей.

**Решение:** Выделите эту общую часть в свой собственный интерфейс.

## § Свёртывание иерархии

**Проблема:** У вас есть некая иерархия классов, в которой подкласс мало чем отличается от суперкласса.

**Решение:** Слейте подкласс и суперкласс воедино.

## § Создание шаблонного метода

**Проблема:** В подклассах реализованы алгоритмы, содержащие похожие шаги и одинаковый порядок выполнения этих шагов.

**Решение:** Вынесите структуру алгоритма и одинаковые шаги в суперкласс, а в подклассах оставьте реализацию отличающихся шагов.

## § Замена наследования делегированием

**Проблема:** У вас есть подкласс, который использует только часть методов суперкласса или не хочет наследовать его данные.

**Решение:** Создайте поле и поместите в него объект суперкласса, делегируйте выполнение методов объекту-суперклассу, уберите наследование.

## § Замена делегирования наследованием

**Проблема:** Класс содержит множество простых делегирующих методов ко всем методам другого класса.

**Решение:** Сделайте класс наследником делегата, после чего делегирующие методы потеряют смысл.

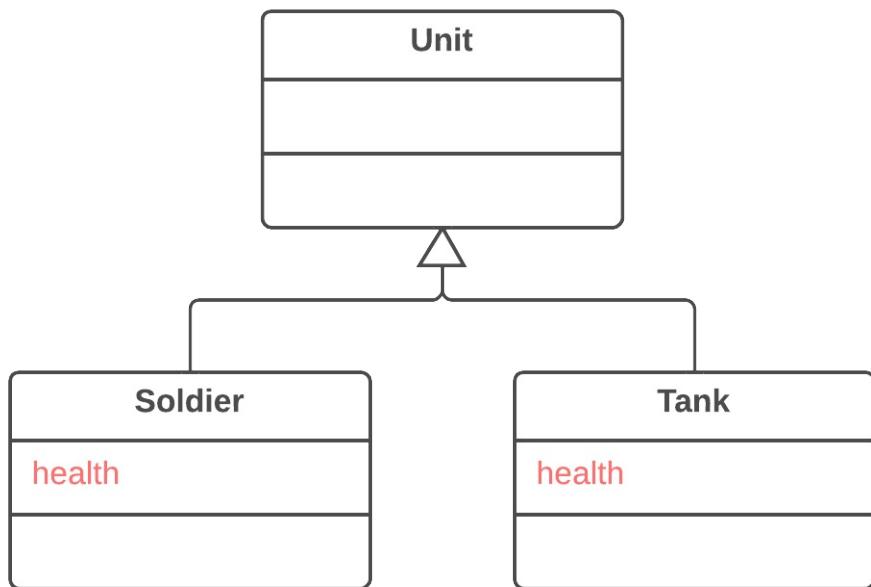


# ✂ Подъём поля

Также известен как *Pull Up Field*

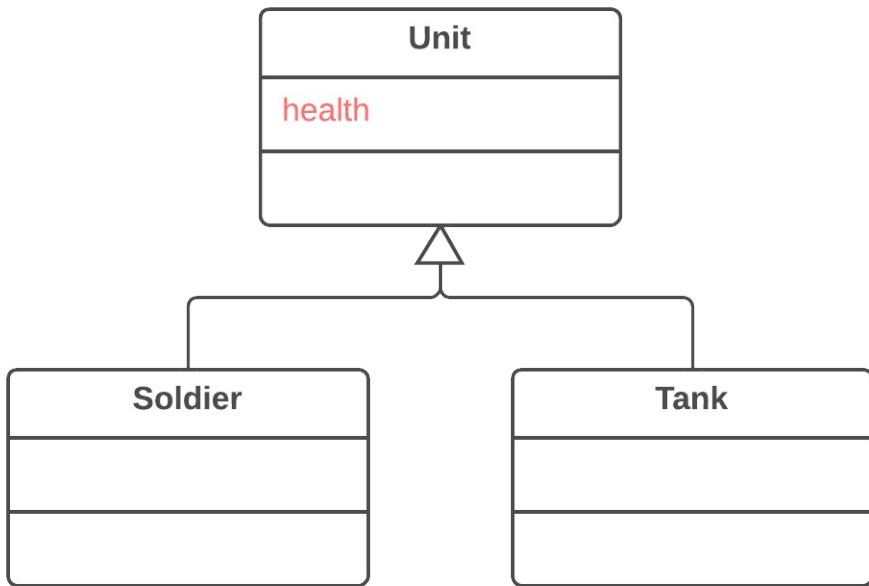
## Проблема

Два класса имеют одно и то же поле.



## Решение

Переместите поле в суперкласс, убрав его из подклассов.



## Причины рефакторинга

Подклассы развивались независимо друг от друга. Это привело к созданию одинаковых (или очень похожих) полей и методов.

## Достоинства

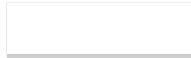
- Убивает дублирование полей в подклассах.
- Облегчает дальнейший перенос дублирующих методов из подклассов в суперкласс, если они есть.

## Порядок рефакторинга

1. Проверьте, что оба поля используются для одинаковых нужд в подклассах.
2. Если поля имеют разные названия, дайте им общее имя и замените все обращения к полям в существующем коде.
3. Создайте поле с таким же именем в суперклассе. Обратите внимание на то, что если поля были приватные (private), поле в суперклассе должно быть защищённым (protected).
4. Удалите поля из подклассов.
5. Возможно, имеет смысл использовать самоинкапсуляцию поля для нового поля, чтобы скрыть его за методами доступа.

## Анти-рефакторинг

§ Спуск поля



## Родственные рефакторинги

§ Подъём метода



## Борется с запахом

§ Дублирование кода



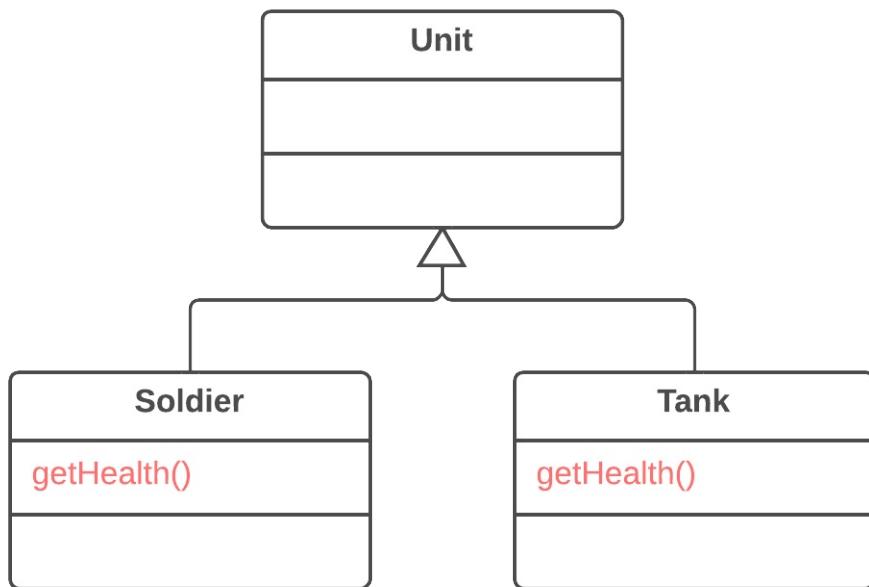


# ✂ Подъём метода

Также известен как *Pull Up Method*

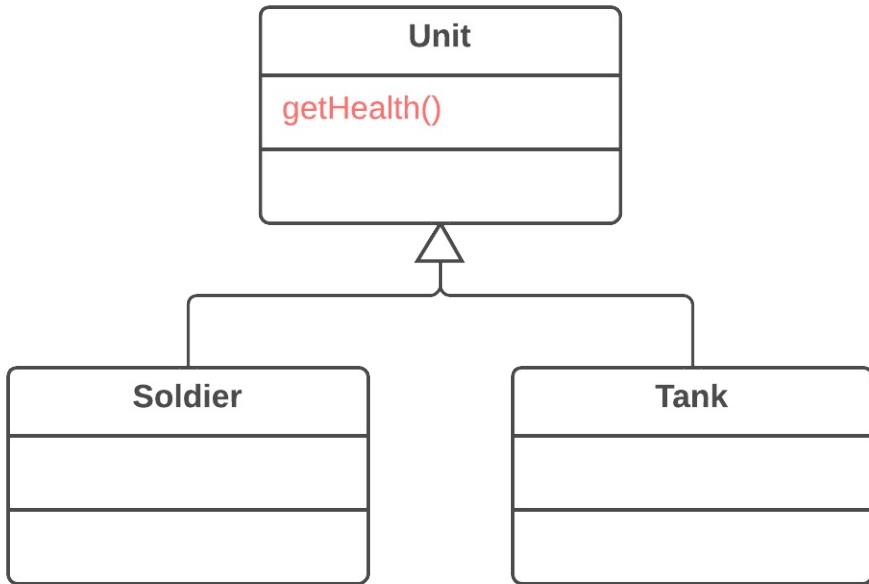
## Проблема

Подклассы имеют методы, которые делают схожую работу.



## Решение

В этом случае нужно сделать методы идентичными, а затем переместить их в суперкласс.



## Причины рефакторинга

Подклассы развивались независимо друг от друга. Это привело к созданию одинаковых (или очень похожих) полей и методов.

## Достоинства

- Убирает дублирование кода. Если вам нужно внести изменения в метод, лучше сделать это в одном месте, чем искать все дубликаты этого метода в подклассах.
- Также этот рефакторинг можно использовать и в случае, если подкласс зачем-то переопределяет метод суперкласса, но, по сути, делает ту же работу.

## Порядок рефакторинга

1. Обследовать похожие методы в суперклассах. Если они не одинаковы, привести их к одному и тому же виду.
2. Если методы используют разный набор параметров, привести эти параметры к тому виду, который вы хотите видеть в суперклассе.
3. Скопируйте метод в суперкласс. Здесь вы можете столкнуться с тем, что код метода использует поля и методы, которые есть только в подклассах, а посему недоступны в суперклассе. Чтобы решить эту проблему, вам нужно:
  - Для полей: либо **поднимите нужные поля в суперкласс**, либо используйте **самоинкапсуляцию поля** для создания геттеров и сеттеров в подклассах, а

затем объявили эти геттеры абстрактным методом в суперклассе.

- Для методов: либо поднимите нужные методы в суперкласс, либо объявили для них абстрактные методы в суперклассе (обратите внимание, ваш класс станет абстрактным, если не был таким до этого).
4. Удалите методы в подклассах.
  5. Проверьте места, в которых вызывается метод. Возможно, в некоторых из них использование подкласса можно заменить суперклассом.

## Анти-рефакторинг

§ Спуск метода



## Родственные рефакторинги

§ Подъём поля



## Помогает рефакторингу

§ Создание шаблонного метода



## Борется с запахом

§ Дублирование кода





# ✖ Подъём тела конструктора

Также известен как *Pull Up Constructor Body*

## Проблема

Подклассы имеют конструкторы с преимущественно одинаковым кодом.

```
class Manager extends Employee {  
  
    public Manager(String name, String id, int grade) {  
  
        this.name = name;  
  
        this.id = id;  
  
        this.grade = grade;  
  
    }  
  
    //...  
  
}
```

## Решение

Создайте конструктор в суперклассе и вынесите в него общий для подклассов код. Вызывайте конструктор суперкласса в конструкторах подкласса.

```
class Manager extends Employee {  
  
    public Manager(String name, String id, int grade) {  
  
        super(name, id);  
  
        this.grade = grade;  
  
    }  
  
    //...  
  
}
```

## Причины рефакторинга

Чем этот рефакторинг отличается от подъёма метода?

- L. В Java подклассы не могут наследовать конструктор, поэтому вы не можете просто применить подъём метода к конструктору подкласса и удалить его после перемещения всего кода конструктора в суперкласс. Вдобавок к созданию конструктора в суперклассе нужно будет иметь конструкторы в подклассах с простым делегированием к конструктору суперкласса.

2. В C++ и Java (в случае, если вы явно не вызвали конструктор суперкласса) конструктор суперкласса автоматически вызывается перед конструктором подкласса, что делает обязательным перемещение общего кода только из начала конструкторов подклассов (т.к. вы не сможете вызвать конструктор суперкласса в произвольном месте конструктора подкласса).
3. В большинстве языков программирования конструктор подкласса может иметь свой собственный список параметров, отличный от параметров суперкласса, поэтому вы должны создать конструктор суперкласса только с теми параметрами, которые ему действительно нужны.

## Порядок рефакторинга

1. Создайте конструктор в суперклассе.
2. Извлеките общий код из начала конструктора каждого из подклассов в конструктор суперкласса. Перед этим действием стоит попробовать поместить как можно больше общего кода в начало конструктора.
3. Поместите вызов конструктора суперкласса первой строкой в конструкторах подклассов.

## Родственные рефакторинги

### § Подъём метода



## Борется с запахом

### § Дублирование кода



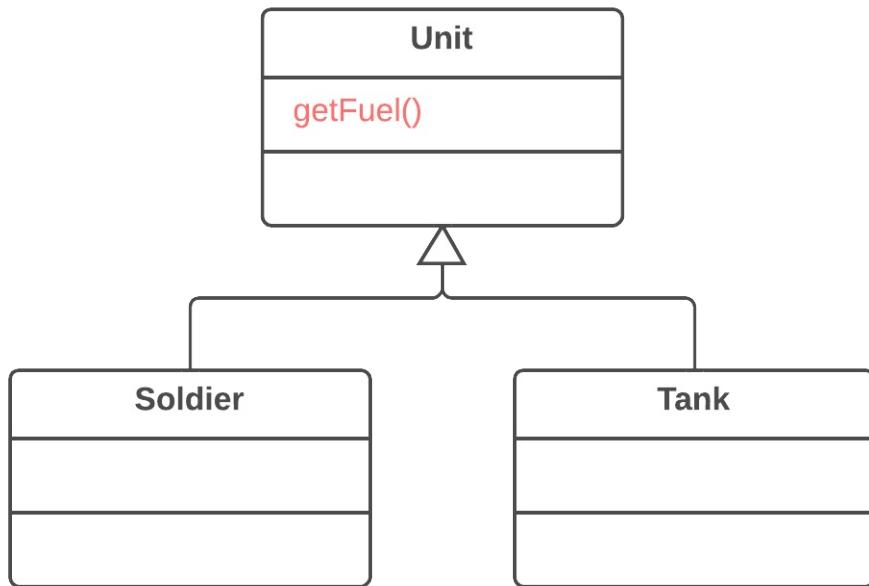


# ✂ Спуск метода

Также известен как Push Down Method

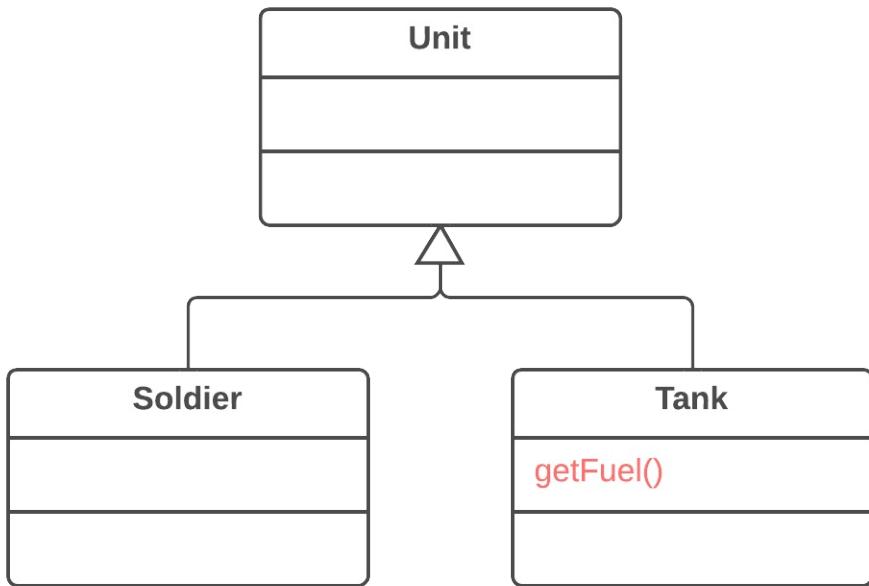
## Проблема

Поведение, реализованное в суперклассе, используется только одним или несколькими подклассами.



## Решение

Переместите это поведение в подклассы.



## Причины рефакторинга

Метод, который планировали сделать универсальным для всех классов, по факту используется только в одном подклассе. Такая ситуация может возникнуть, когда планируемые фичи так и не были реализованы.

Кроме того, такая ситуация может возникнуть после извлечения (или удаления) части функциональности из иерархии классов, после которого метод остался используемым только в одном подклассе.

Если вы видите, что метод необходим более чем одному подклассу (но не всем), возможно, стоит создать промежуточный подкласс и переместить метод в него. Это позволит избежать дублирования кода, которое возникло бы при спуске метода во все подклассы.

## Достоинства

- Улучшает связность внутри классов. Метод находится там, где вы ожидаете его увидеть.

## Порядок рефакторинга

1. Объявите метод в подклассе и скопируйте его код из суперкласса.
2. Удалите метод из суперкласса.
3. Найдите все места, где используется метод, и убедитесь, что он вызывается из нужного подкласса.

# Анти-рефакторинг

§ Подъём метода



## Родственные рефакторинги

§ Спуск поля



## Помогает рефакторингу

§ Извлечение подкласса



## Борется с запахом

§ Отказ от наследства



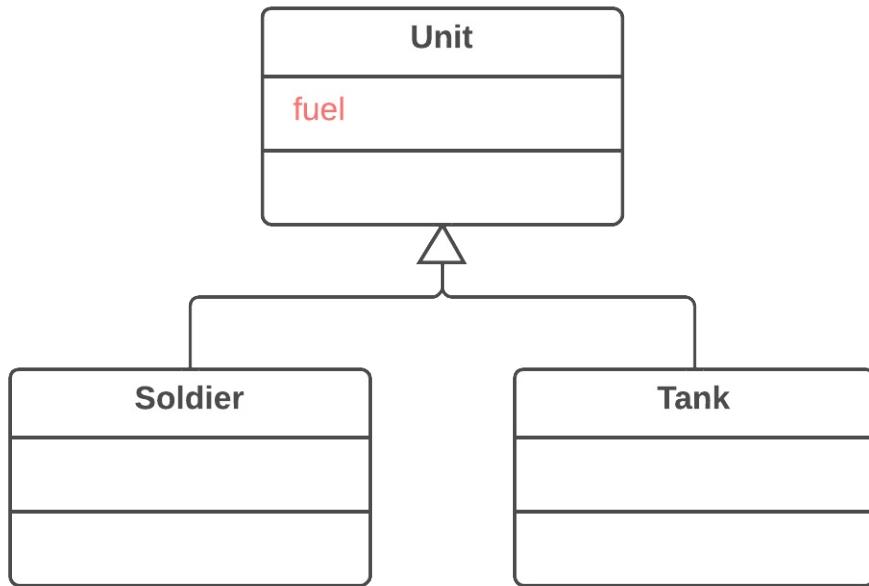


# ✂ Спуск поля

Также известен как *Push Down Field*

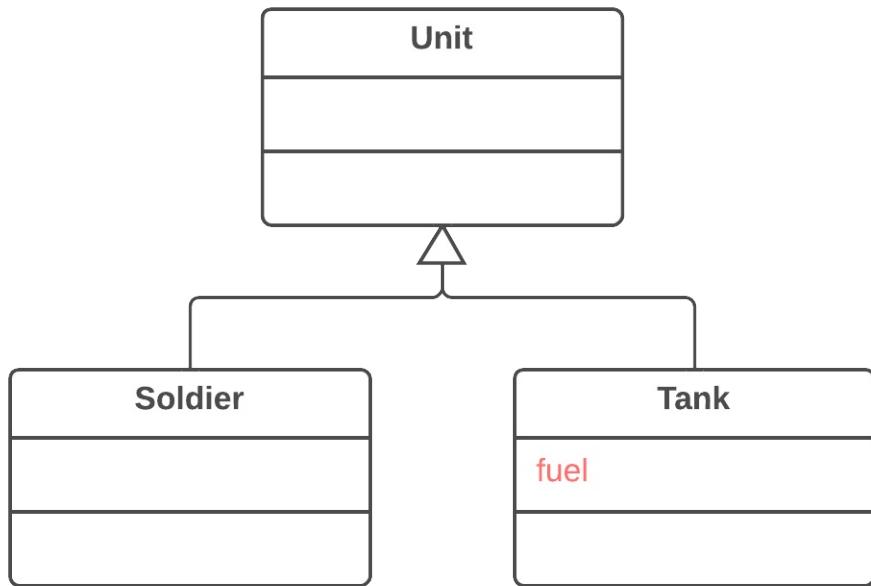
## Проблема

Поле используется только в некоторых подклассах.



## Решение

Переместите поле в эти подклассы.



## Причины рефакторинга

Поле, которое планировали сделать универсальным для всех классов, по факту, используется только в некоторых подклассах. Такая ситуация может возникнуть, когда планируемые фичи так и не были реализованы.

Кроме того, такая ситуация может возникнуть после извлечения (или удаления) части функциональности из иерархии классов.

## Достоинства

- Улучшает связность внутри классов. Поле находится там, где оно реально используется.
- При перемещении в несколько подклассов одновременно, появляется возможность развивать поля независимо друг от друга. Правда, такое действие создаёт дублирование кода, поэтому стоит спускать поля, только если вы действительно намерены использовать их по-разному.

## Порядок рефакторинга

1. Объявите поле во всех необходимых подклассах.
2. Удалите поле из суперкласса.

## Анти-рефакторинг

# Родственные рефакторинги

§ Спуск метода



## Помогает рефакторингу

§ Извлечение подкласса



## Борется с запахом

§ Отказ от наследства



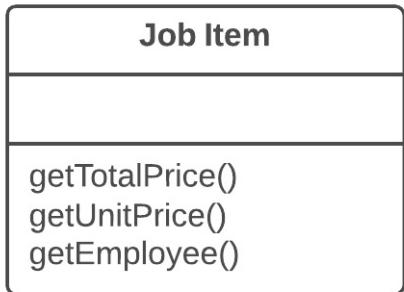


# Извлечение подкласса

Также известен как *Extract Subclass*

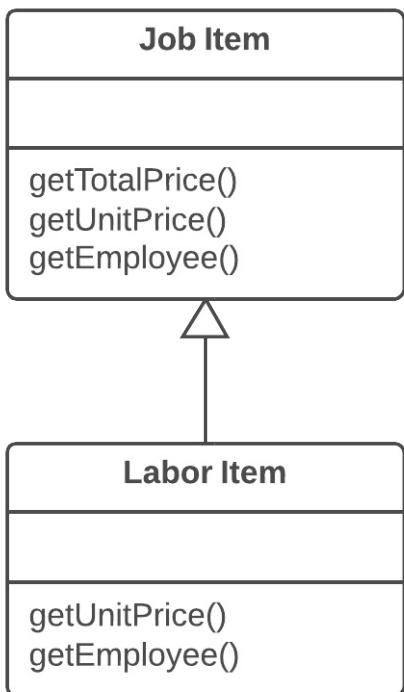
## Проблема

Класс имеет фичи, которые используются только в определённых случаях.



## Решение

Создайте подкласс и используйте его в этих случаях.



## Причины рефакторинга

В основном классе находятся методы и поля для реализации какого-то редкого случая использования класса. Этот случай, хоть и очень редкий, является частью обязанностей класса, поэтому все связанные с ним поля и методы неправильно было бы вынести в абсолютно другой класс. Но, с другой стороны, их можно вынести в подкласс с помощью этого рефакторинга.

## Достоинства

- Создать подкласс довольно легко и быстро.
- Можно выделить несколько разных подклассов, если основной класс реализует несколько подобных особых случаев.

## Недостатки

- Несмотря на всю очевидную простоту, *Наследование* может завести вас в тупик, если придётся выделить несколько различных иерархий классов. Например, если у вас был класс `Собаки`, поведение которого отличается в зависимости от размера и длины шерсти собак. Вы выделили из него две иерархии:
  - по размеру: `Большие`, `Средние` И `Маленькие`
  - по длине шерсти: `Гладкошёрстные` И `Длинношёрстные`
- И все бы хорошо, но у вас начнутся проблемы, когда нужно будет создать одновременно `Большую` И `Гладкошёрстную` собаку, т.к. объект можно создать лишь из одного класса. С другой стороны, эту проблему можно обойти, используя *Композицию* вместо *Наследования* (см. паттерн *Стратегия*). Другими словами, класс `Собака` будет иметь два поля-компоненты – размер и длина шерсти. В эти поля вы будете подставлять объекты-компоненты необходимых классов. Например, вы сможете создать `Собаку`, имеющую `БольшойРазмер` И `ДлиннуюШерсть`.

## Порядок рефакторинга

1. Создайте новый подкласс из интересующего вас класса.
2. Если для создания объектов из подкласса будут нужны какие-то дополнительные данные, создайте конструктор и дополните его нужными параметрами. Не забудьте вызвать родительскую реализацию конструктора.
3. Найдите все вызовы конструктора родительского класса. В тех случаях, когда требуется функциональность подкласса, замените родительский конструктор конструктором подкласса.

- Переместите нужные методы и поля из родительского класса в подкласс. Используйте для этого спуск метода и спуск поля. Пуще всего начинать перенос с методов. Так поля будут доступны для них все время: из родительского класса до переноса, и из самого подкласса после окончания переноса.
- После того как подкласс готов, найдите все старые поля, которые управляли тем, какой набор функций должен выполняться. Эти поля можно удалить, заменив полиморфизмом все условные операторы, в которых они использовались. Простой пример – у вас в классе Автомобиль было поле `isElectricCar`, и в зависимости от него, в методе `refuel()` в машину либо заливается бензин, либо заряжается электричество. В результате рефакторинга, поле `isElectricCar` будет удалено, а классы Автомобиль и ЭлектроАвтомобиль будут иметь свои реализации метода `refuel()`.

## Родственные рефакторинги

### § Извлечение класса



## Борется с запахом

### § Большой класс



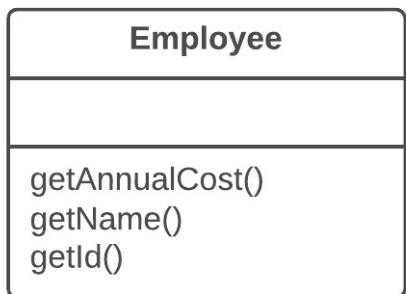
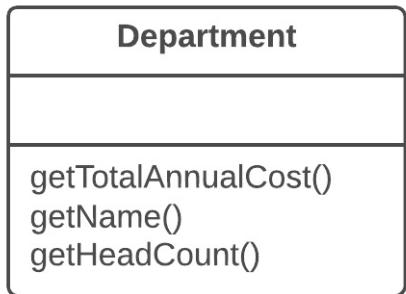


# Извлечение суперкласса

Также известен как *Extract Superclass*

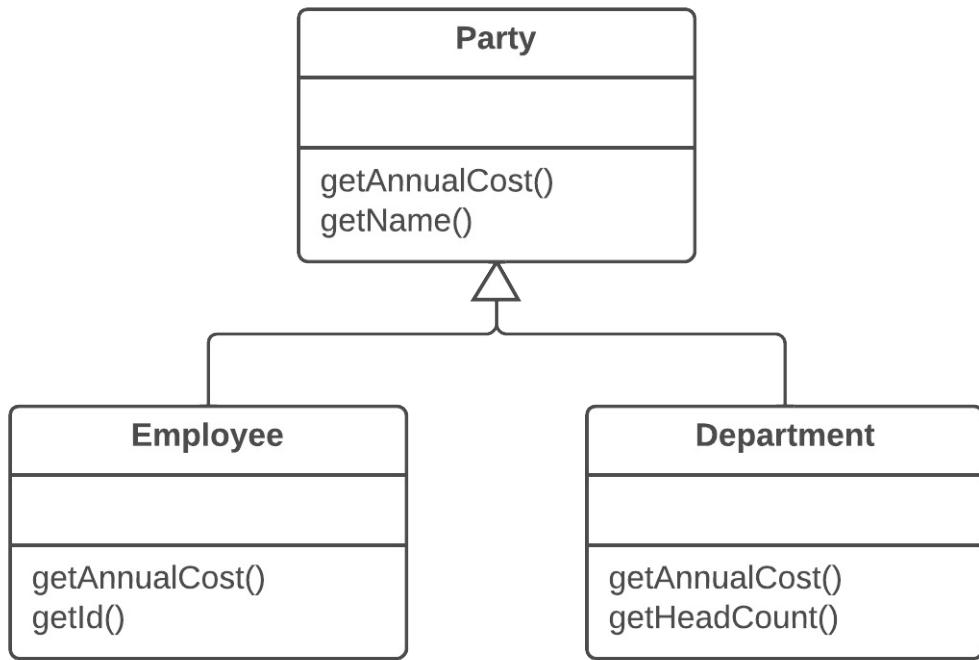
## Проблема

У вас есть два класса с общими полями и методами.



## Решение

Создайте для них общий суперкласс и перенесите туда одинаковые поля и методы.



## Причины рефакторинга

Одним из видов дублирования кода является наличие двух классов, выполняющих сходные задачи одинаковым способом или сходные задачи разными способами. Объекты предоставляют встроенный механизм для упрощения такой ситуации с помощью наследования. Однако часто общность оказывается незамеченной до тех пор, пока не будут созданы какие-то классы, и тогда появляется необходимость создавать структуру наследования позднее.

## Достоинства

- Убирает дублирование кода. Общие поля и методы теперь «живут» только в одном месте.

## Когда нельзя применить

- Вы не можете применить этот рефакторинг к классам, которые уже имеют суперкласс.

## Порядок рефакторинга

1. Создайте абстрактный суперкласс.
2. Используйте подъём поля, подъём метода и подъём тела конструктора для перемещения общей функциональности в суперкласс. Лучше начинать с полей,

т.к. помимо общих полей, вам нужно будет перенести те из них, которые используются в общих методах.

3. Стоит поискать места в клиентском коде, в которых можно заменить использование подклассов вашим общим классом (например, в объявлениях типов).

## Родственные рефакторинги

### § Извлечение интерфейса

## Борется с запахом

### § Дублирование кода

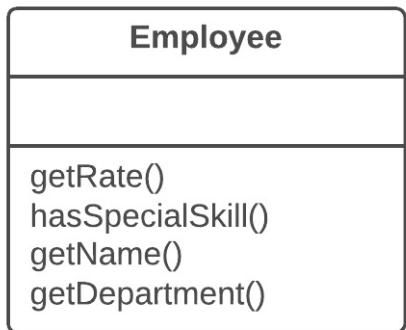


# ✖ Извлечение интерфейса

Также известен как *Extract Interface*

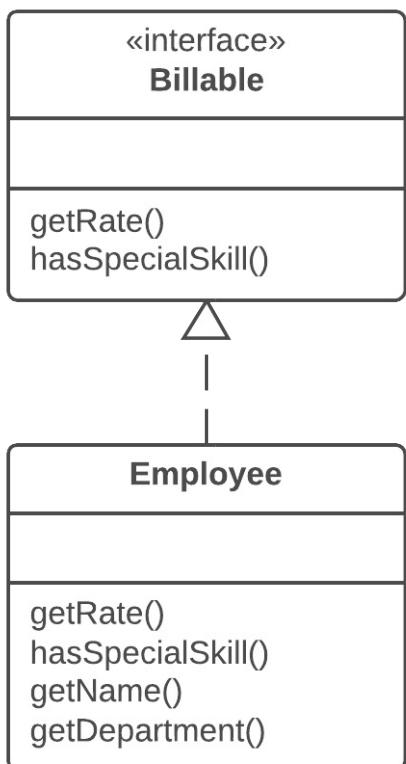
## Проблема

Несколько клиентов пользуются одной и той же частью интерфейса класса. Либо в двух классах часть интерфейса оказалась общей.



## Решение

Выделите эту общую часть в свой собственный интерфейс.



## Причины рефакторинга

1. Интерфейсы бывают кстати, когда один и тот же класс может отыгрывать различные роли в различных ситуациях. Используйте извлечение интерфейса, чтобы явно обозначить каждую из ролей.
2. Ещё один удобный случай возникает, когда требуется описать операции, которые класс выполняет на своём сервере. Если в будущем предполагается разрешить использование серверов нескольких видов, все они должны реализовывать этот интерфейс.

## Полезные факты

Есть некоторое сходство между извлечением суперкласса и извлечением интерфейса.

Извлечение интерфейса позволяет выделять только общие интерфейсы, но не общий код. Другими словами, если в классах находится дублирующий код, то, применив извлечение интерфейса, вы никак не избавитесь от этого дублирования.

Тем не менее, эту проблему можно уменьшить, применив извлечение класса для помещения поведения, содержащего дублирование в отдельный компонент и делегирования ему всей работы. В случае если объем общего поведения окажется довольно большим, всегда можно применить извлечение суперкласса. Конечно, это даже проще, но помните, что при этом вы получаете только один родительский класс.

## Порядок рефакторинга

1. Создайте пустой интерфейс.
2. Объявите общие операции в интерфейсе.
3. Объявите нужные классы как реализующие этот интерфейс.
4. Измените объявление типов в клиентском коде так, чтобы они использовали новый интерфейс.

## Родственные рефакторинги

### § Извлечение суперкласса

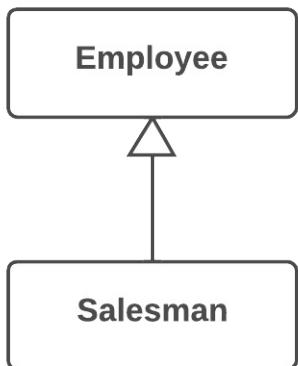


# ✂ Свёртывание иерархии

Также известен как *Collapse Hierarchy*

## Проблема

У вас есть некая иерархия классов, в которой подкласс мало чем отличается от суперкласса.



## Решение

Слейте подкласс и суперкласс воедино.



## Причины рефакторинга

Развитие программы привело к тому, что подкласс и суперкласс стали очень мало отличаться друг от друга. Какая-то фича была убрана из подкласса, какой-то метод «переехал» в суперкласс, и вот вы уже имеете два практически одинаковых класса.

## Достоинства

- Уменьшается сложность программы. Меньше классов, меньше вещей, которые нужно держать в голове, меньше «движущихся частей», меньше вероятность

сломать что-то при последующих изменениях в коде.

- Навигация по коду становится проще, когда методы определены только в одном классе. Нужный метод не приходится искать по всей иерархии.

## Когда нельзя применить

- Если в иерархии классов находится больше одного подкласса, то после проведения рефакторинга, оставшиеся подклассы должны стать наследниками класса, в котором была объединена иерархия.
- Однако имейте в виду, что это может привести к нарушению *принципа подстановки Барбары Лисков*. Например, если в программе эмуляторе городского транспорта неверно свернуть суперкласс `Транспорт` в подкласс `Автомобиль`, класс `Самолёт` может оказаться наследником `Автомобиля`, а это уже неправильно.

## Порядок рефакторинга

1. Выберите, какой класс убрать удобнее: суперкласс или подкласс.
2. Используйте подъём поля и подъём метода, если вы решили избавиться от подкласса. Используйте спуск поля и спуск метода, если убран будет суперкласс.
3. Замените все использование класса, который будет удалён, классом, в который переезжают поля и методы. Зачастую это будет код создания классов, указания типов параметров и переменных, а также документации в комментариях.
4. Удалите пустой класс.

## Родственные рефакторинги

### § Встраивание класса

Свёртывание иерархии является вариантом встраивания класса, где все фичи переезжают в суперкласс или подкласс.

## Борется с запахом

### § Ленивый класс

### § Теоретическая общность

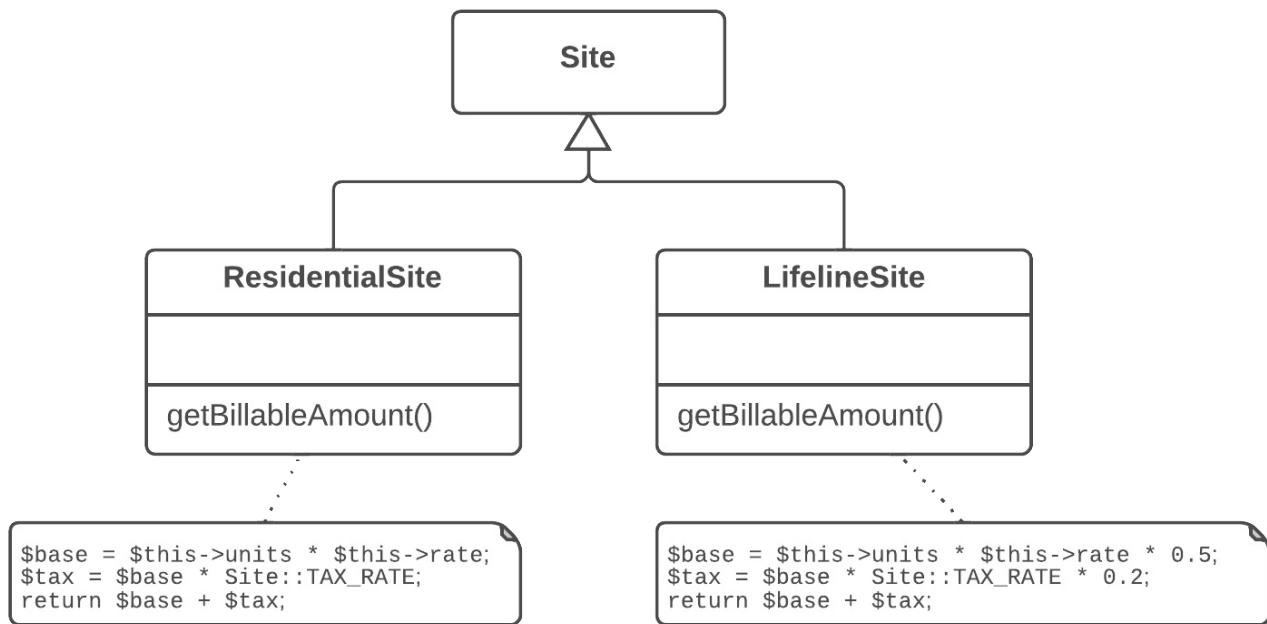


# ❖ Создание шаблонного метода

Также известен как *Form Template Method*

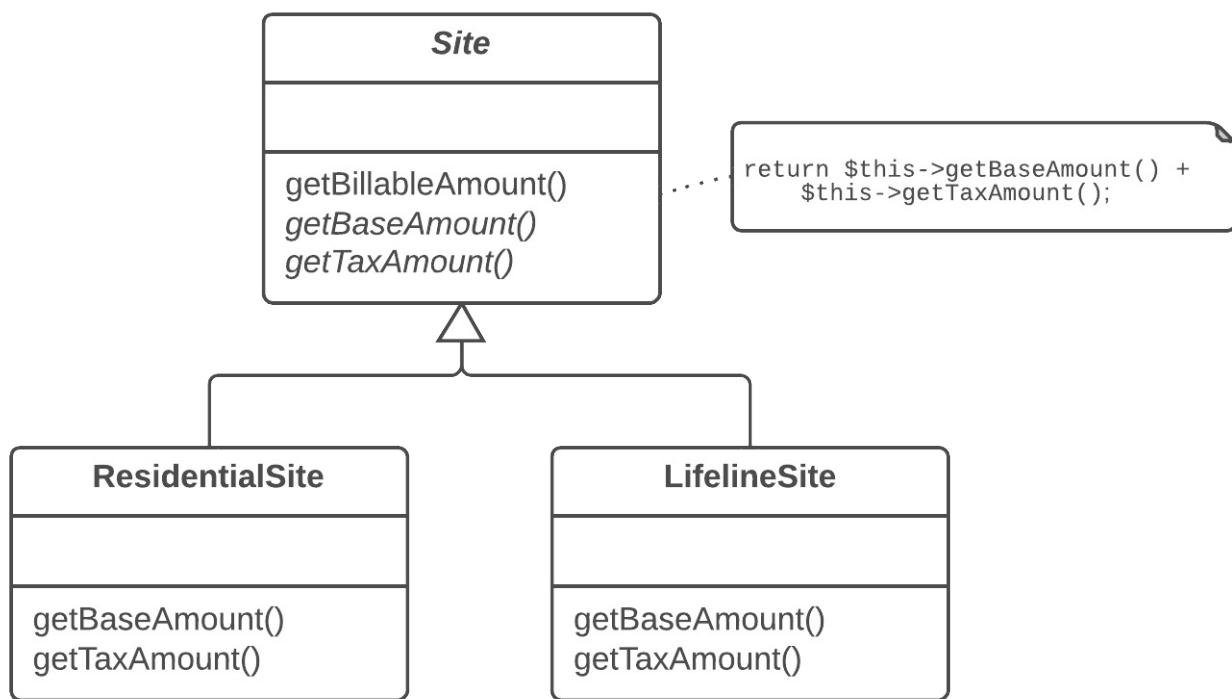
## Проблема

В подклассах реализованы алгоритмы, содержащие похожие шаги и одинаковый порядок выполнения этих шагов.



## Решение

Вынесите структуру алгоритма и одинаковые шаги в суперкласс, а в подклассах оставьте реализацию отличающихся шагов.



## Причины рефакторинга

Подклассы развиваются параллельно. Иногда разными людьми, что приводит к дублированию кода и ошибок, а также к усложнению поддержки, т.к. каждое изменение приходится проводить во всех подклассах.

## Достоинства

- Когда мы говорим о дублировании кода, не всегда имеется в виду программирование методом копирования-вставки. Нередко дублирование возникает на более абстрактном уровне. Например, у вас есть метод сортировки чисел и метод сортировки коллекции объектов, при этом, единственное, чем они отличаются это сравнение элементов. Создание шаблонного метода позволяет справиться с таким дублированием, объединив общие шаги алгоритма в суперклассе и оставив различия для подклассов.
- Создание шаблонного метода реализует *принцип открытости/закрытости*. При появлении новой версии алгоритма, вам нужно будет всего лишь создать новый подкласс, не меняя существующий код.

## Порядок рефакторинга

- Разбейте алгоритмы в подклассах на составные части, описанные в отдельных методах. В этом может помочь извлечение метода.

2. Получившиеся методы, одинаковые для всех подклассов, можете смело перемещать в суперкласс, используя **подъём метода**. [ ]
3. Отличающиеся методы приведите к единым названиям с помощью **переименования метода**. [ ]
4. Поместите сигнатуры отличающихся методов в суперкласс как абстрактные с помощью **подъёма метода**. Их реализации оставьте в подклассах.
5. И наконец, поднимите основной метод алгоритма в суперкласс. Он теперь должен работать с методами-шагами, описанными в суперклассе – реальными или абстрактными.

## Реализует паттерн проектирования

§ **Шаблонный метод** [ ]

## Борется с запахом

§ **Дублирование кода** [ ]

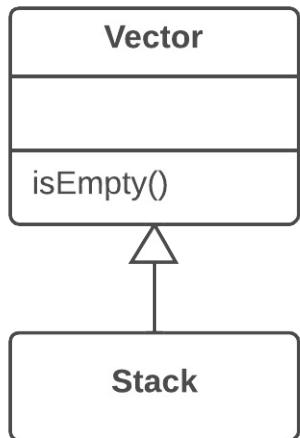


# ☒ Замена наследования делегированием

Также известен как *Replace Inheritance with Delegation*

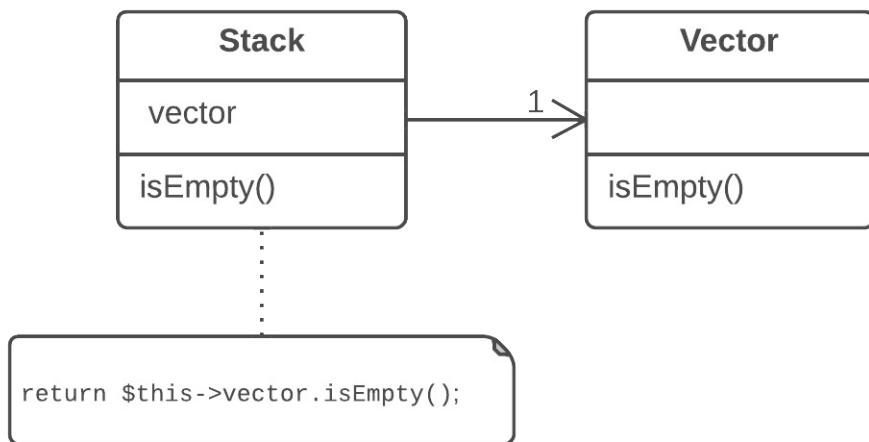
## Проблема

У вас есть подкласс, который использует только часть методов суперкласса или не хочет наследовать его данные.



## Решение

Создайте поле и поместите в него объект суперкласса, делегируйте выполнение методов объекту-суперклассу, уберите наследование.



## Причины рефакторинга

Замена наследования композицией может значительно улучшить дизайн классов, если:

- Ваш подкласс нарушает *принцип замещения Барбары Лисков*. Другими словами, наследование возникло только ради объединения общего кода, но не потому, что подкласс «является» (*is-a*) расширением суперкласса.
- Подкласс использует только часть методов суперкласса. В этом случае, это только вопрос времени, пока кто-то не вызовет метод суперкласса, который он не должен был вызывать.

Суть рефакторинга сводится к тому, чтобы разделить оба класса, и сделать суперкласс помощником подкласса, а не его родителем. Вместо того чтобы наследовать все методы суперкласса, подкласс будет иметь только необходимые методы, которые будут делегировать выполнение методам объекта-суперкласса.

## Достоинства

- Класс не содержит лишних методов, которые достались ему в наследство от суперкласса.
- В поле-делегат можно подставлять разные объекты, имеющие различные реализации функциональности. По сути, вы получаете реализацию паттерна проектирования **Стратегия**.



## Недостатки

- Приходится писать очень много простых делегирующих методов.

## Порядок рефакторинга

1. Создайте поле в подклассе для содержания суперкласса. На первом этапе поместите в него текущий объект.
2. Измените методы подкласса так, чтобы они использовали объект суперкласса, вместо `this`.
3. Для методов, которые были унаследованы из суперкласса и которые вызывается в клиентском коде, в подклассе нужно создать простые делегирующие методы.
4. Уберите объявление наследования из подкласса.
5. Измените код инициализации поля, в котором хранится бывший суперкласс, созданием нового объекта.

## Анти-рефакторинг

§ Замена делегирования наследованием

---

## Реализует паттерн проектирования

§ Стратегия



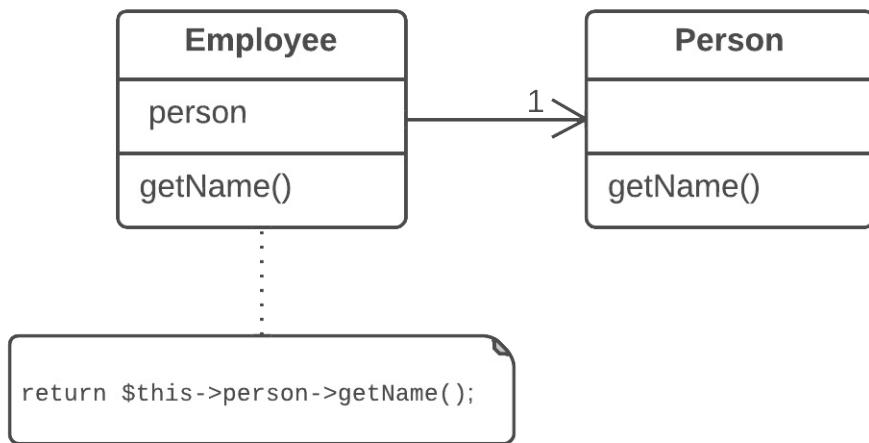


# ☒ Замена делегирования наследованием

Также известен как *Replace Delegation with Inheritance*

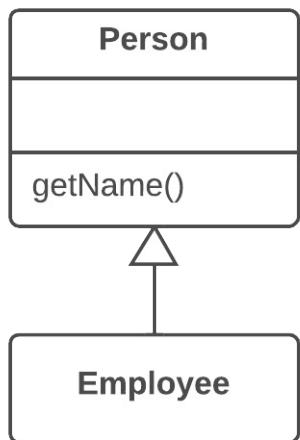
## Проблема

Класс содержит множество простых делегирующих методов ко всем методам другого класса.



## Решение

Сделайте класс наследником делегата, после чего делегирующие методы потеряют смысл.



## Причины рефакторинга

Делегирование является более гибким подходом, чем наследование, т.к. позволяет изменять поведение класса на лету, заменяя объект к которому происходит делегирование. Тем не менее, применение делегирования перестаёт быть выгодным, если вы делегируете действия только одному классу, причём всем его публичным методам.

Если в этом случае заменить делегирование наследованием, вы избавите класс от множества делегирующих методов, а себя от необходимости создавать их для каждого нового метода класса-делегата.

## Достоинства

- Уменьшает количество кода. Вам больше не нужны все эти делегирующие методы.

## Когда нельзя применить

- Не применяйте рефакторинг, если класс содержит делегирование только к части публичных методов класса-делегата. Этим вы нарушите *принцип замещения Барбары Лисков*.
- Этот рефакторинг может быть применён только если класс ещё не имеет родителей.

## Порядок рефакторинга

1. Сделайте класс подклассом класса-делегата.
2. В поле, содержащее ссылку на объект-делегат, поставьте текущий объект.
3. Один за другим удаляйте методы с простым делегированием. Если у них отличались названия, используйте переименование метода чтобы привести все методы к одному названию.
4. Замените все обращения к полю-делегату обращениями к текущему объекту.
5. Удалите поле-делегат.

## Анти-рефакторинг

### § Замена наследования делегированием

## Родственные рефакторинги

### § Удаление посредника

# **Борется с запахом**

## **§ Неуместная близость**





# Послесловие

## Поздравляю! Вы добрались до конца!

Чем теперь займёшься?

Пойдёте рефакторить один из ваших старых проектов? Или, может, похвастаетесь своими новыми супер-способностями перед друзьями и коллегами, сделав код-ревью текущего проекта?

В любом случае, вот вам парочка идей для следующих шагов, если вы ещё не определились:

- Прочитайте книгу Джошуа Кериевски «[Рефакторинг с использованием паттернов проектирования](#)».
- Плаваете в паттернах? [Подучите матчасть](#).
- Распечатайте [шпаргалки по рефакторингу](#) и повесьте их где-то на видном месте.
- [Оставьте отзыв](#) об этой книге и [полном курсе](#). Мне было бы очень интересно услышать ваше мнение, даже если это критика