**Рефакторинг** – процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание ее работы.

1. **Цель рефакторинга** – упростить понимание и модификацию программного обеспечения.
2. Рефакторинг не меняет видимого поведения программного обеспечения.

**Причины применения Рефакторинга:**

1. необходимо добавить новую функцию, которая недостаточно укладывается в принятое [архитектурное решение](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F);
2. необходимо исправить ошибку, причины возникновения которой сразу не ясны;
3. преодоление трудностей в командной разработке, которые обусловлены сложной логикой программы.

**Признаки плохого кода:**

1. [дублирование кода](https://ru.wikipedia.org/wiki/%D0%94%D1%83%D0%B1%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BA%D0%BE%D0%B4%D0%B0);
2. длинный метод;
3. большой класс;
4. длинный список параметров;
5. «жадные» функции — это метод, который чрезмерно обращается к данным другого объекта;
6. избыточные временные переменные;
7. классы данных;
8. несгруппированные данные.

**Рефакторинг улучшает композицию программного обеспечения!**

Без рефакторинга композиция программы приходит в негодность. По мере внесения в код изменений, связанных с реализацией краткосрочных целей или производимых без полного понимания организации кода, последний утрачивает свою структурированность. Разобраться в проекте, читая код, становится все труднее. Рефакторинг напоминает наведение порядка в коде. Убираются фрагменты, оказавшиеся не на своем месте. Утрата кодом структурности носит кумулятивный характер. Чем сложнее разобраться во внутреннем устройстве кода, тем труднее его сохранить и тем быстрее происходит его распад. Регулярно проводимый рефакторинг помогает сохранять форму кода. Плохо спроектированный код обычно занимает слишком много места, часто потому, что он выполняет в нескольких местах буквально одно и то же. Поэтому важной стороной улучшения композиции является удаление дублирующегося кода. Важность этого связана с модификациями кода в будущем. Сокращение объема кода не сделает систему более быстрой, потому что изменение размера образа программы в памяти редко имеет значение. Однако объем кода играет существенную роль, когда приходится его модифицировать. Чем больше кода, тем труднее правильно его модифицировать, и разбираться приходится в его большем объеме. При изменении кода в некотором месте система ведет себя несоответственно расчетам, поскольку не модифицирован другой участок, который делает то же самое, но в несколько ином контексте. Устраняя дублирование, мы гарантируем, что в коде есть все, что нужно, и притом только в одном месте, в чем и состоит суть хорошего проектирования.

**Рефакторинг облегчает понимание программного обеспечения!**

Во многих отношениях программирование представляет собой общение с компьютером. Программист пишет код, указывающий компьютеру, что необходимо сделать, и тот в ответ делает в точности то, что ему сказано. Со временем разрыв между тем, что требуется от компьютера, и тем, что вы ему говорите, сокращается. В таком режиме суть программирования состоит в том, чтобы точно сказать, что требуется от компьютера. Но программа адресована не только компьютеру. Пройдет некоторое время, и кому-нибудь понадобится прочесть ваш код, чтобы внести какието изменения.

Рефакторинг помогает сделать код более легким для чтения. При проведении рефакторинга вы берете код, который работает, но не отличается идеальной структурой. Потратив немного времени на рефакторинг, можно добиться того, что код станет лучше информировать о своей цели. (Программисты ленивые, нет комментариев, код написан- работает и слава богу)

В процессе рефакторинга программисты повышают свой уровень знания, начинают лучше понимать язык и в частности программу которую они оптимизируют.

**Рефакторинг помогает найти ошибки!**

Лучшее понимание кода помогает выявить ошибки. Постоянное выявление ошибок, помогает в дальнейшем писать чистый, правильный и надежный код (часто встречающиеся ошибки).

**Когда следует проводить Рефакторинг?**

Правильнее всего проводить рефакторинг постоянно, понемногу.

**Методы Рефакторинга:**

**Изменение сигнатуры метода**

Суть изменения сигнатуры метода заключается в добавлении, изменении или удалении параметра метода (Функция.1,2,3,4,5 имеет числовые атрибуты, мы меняем у функции.1 на текстовый атрибут, в дальнейшем функция 5 запрашивает данные у функции.1 и в результате получает ошибку)

**Инкапсуляция пола**

В случае, если у класса имеется открытое поле, необходимо сделать его закрытым и обеспечить методы доступа.

**Выделение метода**

Выделение метода заключается в выделении из длинного и/или требующего комментариев кода отдельных фрагментов и преобразовании их в отдельные методы, с подстановкой подходящих вызовов в местах использования. В этом случае действует правило: если фрагмент кода требует комментария о том, что он делает, то он должен быть выделен в отдельный метод. Также правило: один метод не должен занимать более чем один экран (25-50 строк, в зависимости от условий редактирования), в противном случае некоторые его фрагменты имеют самостоятельную ценность и подлежат выделению. Из анализа связей выделяемого фрагмента с окружающим контекстом делается вывод о перечне параметров нового метода и его локальных переменных.

**Перемещение метода**

Перемещение метода применяется по отношению к методу, который чаще обращается к другому классу, чем к тому, в котором сам располагается.

**Замена условного оператора полиморфизмом**

[Условный оператор](https://ru.wikipedia.org/wiki/%D0%A3%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80%22%20%5Co%20%22%D0%A3%D1%81%D0%BB%D0%BE%D0%B2%D0%BD%D1%8B%D0%B9%20%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80) с несколькими ветвями заменяется вызовом полиморфного метода некоторого базового класса, имеющего подклассы для каждой ветви исходного оператора. Выбор ветви осуществляется неявно, в зависимости от того, экземпляру какого из подклассов оказался адресован вызов.