Рефакторинг

Что такое рефакторинг?

Рефакторинг представляет собой процесс такого изменения программной системы, при котором не меняется внешнее поведение кода, но улучшается его внутренняя структура. Это способ систематического приведения кода в порядок, при котором шансы появления новых ошибок минимальны. В сущности, при проведении рефакторинга кода вы улучшаете его дизайн уже после того, как он написан.

Правила рефакторинга

- Обнаружив, что в программу необходимо добавить новую функциональность, но код программы не структурирован удобным для добавления этой функциональности образом, сначала произведите рефакторинг программы, чтобы упростить внесение необходимых изменений, а только потом добавьте функцию.
- Перед началом рефакторинга убедитесь, что располагаете надежным комплектом тестов. Эти тесты должны быть самопроверяющимися.
- При применении рефакторинга программа модифицируется небольшими шагами. Ошибку нетрудно обнаружить.
- Написать код, понятный компьютеру, может каждый, но только хорошие программисты пишут код, понятный людям.

Принципы рефакторинга

- **Рефакторинг (Refactoring):** изменение во внутренней структуре программного обеспечения, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения.
- **Производить рефакторинг (Refactor):** изменять структуру программного обеспечения, применяя ряд рефакторингов, не затрагивая его поведения.

Рефакторинг не меняет видимого поведения программного обеспечения. Оно продолжает выполнять прежние функции. Никто — ни конечный пользователь, ни программист — не сможет сказать по внешнему виду, что что-то изменилось.

Когда следует проводить рефакторинг?

- **Правило трех ударов** — делая что-то в первый раз, вы просто это делаете. Делая что-то аналогичное во второй раз, вы морщитесь от необходимости повторения, но все-таки повторяете то же самое. Делая что-то похожее в третий раз, вы начинаете рефакторинг.
- Применяйте рефакторинг при добавлении новой функции
- Применяйте рефакторинг, если требуется исправить ошибку
- Применяйте рефакторинг при разборе кода

Как следует проводить рефакторинг?

Рефакторинг следует проводить серией небольших изменений, каждое из которых делает существующий код чуть лучше, оставляя программу в рабочем состоянии.

Рефакторинг проведится правильно, если:

- **Код становится чище.**
- **В процессе рефакторинга не создаётся новая функциональность.**
- **Все существующие тесты после изменений успешно проходятся.**

Признаки того, что нужен рефакторинг

Признаки можно разделить на группы:

- **Раздувальщики**
- **Нарушители объектного дизайна**
- **Утяжелители изменений**
- **Замусориватели**
- **Опутыватели связями**

Далее будут рассмотрены эти группы признаков

Раздувальщики

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

К этой группе относятся такие признаки:

- **Длинный метод**
- **Большой класс**
- **Длинный список параметров**
- **Одинаковые группы данных**
- **Большое количество элементарных типов**

Нарушители объектно-ориентированного дизайна

Все эти признаки являют собой неполное или неправильное использование возможностей объектно-ориентированного программирования.

К этой группе признаков относятся:

- **Сложные операторы switch или последовательности if-ов**
- **Временное поле**
- **Подкласс использует малую часть унаследованных полей и методов**
- **Альтернативные классы с разными интерфейсами**

Утяжелители изменений

Эти признаки заметны тогда, когда при необходимости что-то поменять в одном месте программы, вам приходится вносить множество изменений в других местах.

К этим признакам относятся:

- **Расходящиеся модификации** (При внесении изменений в класс приходится изменять большое число различных методов)
- **Стрельба дробью** (При выполнении любых модификаций приходится вносить множество мелких изменений в большое число классов)
- **Параллельные иерархии наследования**

Замусориватели

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

- **Комментарии**
- **Дублирование кода**
- **Ленивый класс**
- **Класс данных**
- **Мёртвый код**
- **Теоретическая общность** (Класс, метод, поле или параметр не используются)

Опутыватели связями

Все признаки 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**
- **Посредник**

Методы рефакторинга

Методы рефакторинга также можно разделить на несколько основных групп:

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

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 изменнения.

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

Далее будут приведены основные методы рефакторинга из этой группы.

Составление методов

- **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.

Составление методов

- **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.

Составление методов

- **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.

Перемещение функций между объектами

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 функций между объектами

- **Перемещение метода**

Проблема: Метод используется 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.

Перемещение функций между объектами

- **Встраивание класса**

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

Решение: Переместите все функции из описанного класса в другой.

- **Сокрытие делегирования**

Проблема: Клиент получает объект B из поля или метода объекта A. Затем клиент вызывает какой-то метод объекта B.

Решение: Создайте новый метод в классе A, который бы делегировал вызов объекту B. Таким образом, клиент перестанет знать о классе B 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.

Перемещение функций между объектами

- **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.

Организация данных

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

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

Организация данных

- **Самоинкапсуляция поля**

Проблема: Вы используете прямой доступ к приватным полями 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 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.

Упрощение условных выражений

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

Упрощение условных выражений

- **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 одинаковому результату или действию.

Решение: Объедините все условия в одном условном операторе.

- **Удаление управляющего флага**

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

Решение: Используйте break, continue и return вместо этой переменной.

Упрощение условных выражений

- **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.

Упрощение вызовов методов

- **Переименование метода**

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

Решение: Измените название метода.

- **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.

- **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.

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.