Билет 13

1. **Рефакторинг по Фаулеру**

Рефакторинг - изменение во внутренней структуре программного обеспечения, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения.

Производить рефакторинг (Refactor) (глагол) - изменять структуру программного обеспечения, применяя ряд рефакторингов, не затрагивая его поведения.

Цель рефакторинга - упростить понимание и модификацию программного обеспечения. Рефакторингом будут только те изменения, которые сделаны с целью облегчения понимания исходного кода. Противоположным примером может служить оптимизация производительности. Как и рефакторинг, оптимизация производительности обычно не изменяет поведения компонента (за исключением скорости его работы); она лишь изменяет его внутреннее устройство. Оптимизация производительности часто затрудняет понимание кода.

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.

Применение рефакторинга при разработке ПО разделяет время между двумя разными видами деятельности - вводом новых функций и изменением структуры. Добавление новых функций не должно менять структуру существующего кода: просто вводятся новые возможности. Прогресс можно оценить, добавляя тесты и добиваясь их нормальной работы. При проведении рефакторинга вы стремитесь не добавлять функции, а только улучшать структуру кода. При этом не добавляются новые тесты (если только не обнаруживается пропущенная ранее ситуация); тесты изменяются только тогда, когда это абсолютно необходимо, чтобы проверить изменения в интерфейсе.

В процессе разработки ПО может оказаться необходимым часто переключаться между двумя видами работы. Попытавшись добавить новую функцию, можно обнаружить, что это гораздо проще сделать, если изменить структуру кода. Тогда следует на некоторое время переключиться на рефакторинг. Улучшив структуру кода, можно добавлять новую функцию. А добившись её работы, можно заметить, что она написана способом, затрудняющим её понимание, тогда вы снова переключаетесь и занимаетесь рефакторингом. Всё это может происходить в течение десяти минут, но в каждый момент вы должны понимать, которым из видов работы заняты.

Рефакторинг - этакие "серебряные плоскогубцы", позволяющие крепко ухватить код. Рефакторинг - это инструмент, который можно и должно использовать для нескольких целей.

Без рефакторинга композиция программы приходит в негодность. Он напоминает наведение порядка в коде. Убираются фрагменты, оказавшиеся не на своём месте. Чем сложнее разобраться во внутреннем устройстве кода, тем труднее его сохранить и тем быстрее происходит его распад. Регулярно проводимый рефакторинг помогает сохранять форму кода.

Важной стороной улучшения композиции является удаление дублирующегося кода. Объём кода играет существенную роль, когда приходится его модифицировать. Чем больше кода, тем труднее правильно его модифицировать, и разбираться приходится в его большем объёме. Устраняя дублирование, мы гарантируем, что в коде есть всё, что нужно, и притом только в одном месте, в чём и состоит суть хорошего проектирования.

Рефакторинг облегчает понимание ПО. После рефакторинга легче понимать незнакомый код, не нужно запоминать деталей кода, которые вы пишете.

Программист может потратить неделю на модификацию кода, хотя мог он разобраться в коде, он бы потратил на это один час.

Рефакторинг помогает найти ошибки. При проведении рефакторинга легче вникать в код, понять, что он делает, и достигнутое понимание возвращается обратно в код.

Рефакторинг позволяет быстрее писать программы. Рефакторинг способствует ускорению разработки кода. Без рефакторинга время будет тратиться не на добавление новых функций, а на поиск и исправление ошибок.

Хороший дизайн нужен для сохранения скорости разработки программного обеспечения. Благодаря рефакторингу программы разрабатываются быстрее, так как он удерживает композицию системы от распада. С его помощью можно даже улучшить дизайн.

Рефакторингом стоит заниматься постоянно понемногу.

Правило трёх ударов от Дона Робертса: "Делая что-то в первый раз, вы просто это делаете. Делая что-то аналогичное во второй раз, вы морщитесь от необходимости повторения, но всё-таки повторяете то же самое. Делая что-то похожее в третий раз, вы начинаете рефакторинг".

Мой код может казаться понятным мне, но не моей команде. Разборы кода знания становятся достоянием всей команды разработчиков. Разборы помогают большему числу людей разобраться с большим числом аспектов крупной программной системы. Разборы также дают возможность большему числу людей высказать полезные мысли. Рефакторинг помогает разобраться в коде, написанном не вами.

Ценность программы имеет две составляющие - то, что они могут делать для нас сегодня и то, что они смогут делать завтра. В большинстве случаев сосредотачиваются только на сегодня.

Но выполнение задачи сегодня не гарантирует выполнение новой задачи завтра. Рефакторинг - путь к решению данной проблемы.

Нам нужны программы, которые легко читать, вся логика которых задана в одном и только одном месте, модификация которых не ставит под угрозу существующие функции и которые позволяют выражать условную логиу возможно более простым способом.

Рефакторинг представляет собой процесс улучшения работающей программы не путём изменения её функций, а путем усиления в ней указанных качеств, позволяющих продолжить разработку с высокой скоростью.

Учитывая одержимость разработчиков программного обеспечения косвенностью, не следует удивляться тому, что рефакторинг, как правило, вводит в программу дополнительную косвенность. Рефакторинг обычно разделяет большие объекты, как и большие методы, на несколько меньших.

Однако рефакторинг - меч обоюдоострый. Каждый раз при разделении чего-либо надвое количество объектов управления растёт. При этом может также быть затруднено чтение программы, потому что один объект делегирует полномочия другому, который делегирует их третьему. Поэтому желательно минимизировать косвенность.

Но не следует спешить, потому что косвенность может окупиться следующими способами:

- позволить совместно использовать логику. Например, подметод, вызываемый из разных мест, или метод родительского класса, доступный всем подметодам.

- изолировать изменения в коде. Допустим, я использую объект в двух разных местах и мне надо изменить его поведение в одном из этих двух случаев. Если изменить объект, это может повлиять на оба случая, поэтому я сначала создаю подкласс и пользуюсь им в том случае, если нужны изменения. В результате можно модифицировать родительский класс без риска непреднамеренного изменения во втором случае.

- кодировать условную логику. В объектах есть сказочный механизм, такой как полиморфные сообщения, гибко, но ясно выражающие условную логику. Преобразовав явные условные операторы в сообщения, часто можно одновременно уменьшить дублирование, улучшить понятность и увеличить гибкость.

Применение рефакторинга исключает возможность испортить систему. После него программа ведёт себя так же, как и прежде. Сверх того, есть возможность добавить в код ценные качества.

Не публикуйте интерфейсы раньше срока. Измените политику в отношении владения кодом, чтобы облегчить рефакторинг.

В некоторых случаях рефакторинг совсем не нужен. Основной пример - необходимость переписать программу с нуля, так как она полностью неработоспособна и уже не фиксится. Второй пример - близость даты завершения проекта. Потому что рост производительности, достигаемый благодаря рефакторингу, проявит себя слишком поздно - после истечения срока. Уорд Каннингем сравнивает незавершённый рефакторинг с залезанием в долги.

Существует утверждение, что рефакторинг может быть альтернативой предварительному проектированию. В таком сценарии проектирование вообще отсутствует. Первое решение, которое приходит в голову, воплощается в коде, а затем доводится до рабочего состояния и обретает требуемую форму с помощью рефакторинга. "Экстремальные" программисты любят такой подход.

Но такой подход не является эффективным, и даже "экстремальные" программисты сначала разрабатывают некую архитектуру будущей системы. Они пробуют разные идеи с помощью CRC-карт или чего-либо подобного, пока не получат внушающего доверия первоначального решения. Только после первого более или менее удачного "выстрела" приступают к кодированию, а затем и к рефакторингу. Смысл в том, что при использовании рефакторинга изменяется роль предварительного проектирования. Если не рассчитывать на рефакторинг, то ощущается необходимость как можно лучше провести предварительное проектирование. Возникает чувство, что любые изменения в проект в будущем, если они потребуются, окажутся слишком дорогостоящими.

С применением рефакторинга акценты смещаются. Предварительное проектирование сохраняется, но теперь оно не имеет целью найти единственное правильное решение. Всё, что от него требуется, - это найти приемлемое решение.

Рефакторинг позволяет создавать более простые проекты, не жертвуя гибкостью, благодаря чему процесс проектирования становится более лёгким и менее напряжённым. Научившись в целом распознавать то, что легко поддаётся рефакторингу, о гибкости решений даже перестаёшь задумываться. Появляется уверенность в возможности применения рефакторинга, когда это понадобится. Создаются самые простые решения, которые могут работать, а гибкие и сложные решения по большей части не потребуются.

Первыми, кто понял важность рефакторинга, были Уорд Каннингем и Кент Бек, с 1980-х работающих со Smalltalk. У Smalltalk очень короткий цикл "компиляция-компоновка-выполнение", поэтому можно быстро модифицировать программы.

2. **Шаблон проектирования Strategy**



**Название и классификация паттерна**

Стратегия - это паттерн поведения объектов.

**Назначение**

Данный паттерн определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.

Другие названия: Policy (политика).

Мотивация

Существует множество алгоритмов для разбиения текста на строки. Жестко "зашивать" все эти алгоритмы в классы, которые в них нуждаются, нежелательно по нескольким причинам:

- Клиенты, требующие алгоритм разбиения на строки, усложняются при включении соответствующего кода. Это приводит к увеличению сложности клиентов и создает проблемы в сопровождении, особенно если необходимо поддерживать несколько алгоритмов одновременно.

- В различных сценариях может потребоваться использование разных алгоритмов. Поддерживать несколько алгоритмов разбиения на строки, если они не используются, было бы неэффективно.

- Если разбиение на строки - неотъемлемая часть клиента, то задачи добавления новых или модификации существующих алгоритмов становятся более сложными.

Эти проблемы можно избежать, определив классы, которые инкапсулируют различные алгоритмы разбиения на строки. Таким образом инкапсулированный алгоритм называется стратегией.

**Примеры**:

- Класс Composition отвечает за разбиение на строки текста, отображаемого в окне программы просмотра, и его своевременное обновление. Стратегии разбиения на строки определяются не в классе Composition, а в подклассах абстрактного класса Compositor.

- Несколько примеров стратегий:

- SimpleCompositor реализует простую стратегию, выделяющую по одной строке за раз.

- TeXCompositor реализует алгоритм разбиения на строки, используемый в редакторе TeX. Эта стратегия оптимизирует разбиение на строки в целом абзаце.

- ArrayCompositor реализует стратегию расстановки переходов на новую строку таким образом, чтобы в каждой строке было одинаковое количество элементов. Например, это может быть полезно при построчном отображении набора пиктограмм.

Объект Composition содержит ссылку на объект Compositor. Каждый раз, когда объекту Composition требуется переформатировать текст, он делегирует эту обязанность объекту Compositor. Чтобы указать, какой объект Compositor должен использоваться, клиент встраивает его в объект Composition.

**Применимость**

Основные условия для применения паттерна стратегия:

- Наличие нескольких родственных классов, отличающихся только поведением. Стратегия позволяет настроить класс на один из множества возможных вариантов поведения.

- Наличие нескольких вариантов алгоритма, реализованных в виде иерархии классов.

- Алгоритм содержит данные, которые клиент не должен знать. Использование стратегии позволяет скрыть сложные структуры данных, специфичные для алгоритма.

- В классе определено множество вариантов поведения через разветвленные условные операторы. В таком случае код из ветвей можно перенести в отдельные классы стратегий для упрощения.

**Структура**



**Участники**

- Strategy (Compositor) — стратегия:

- объявляет общий интерфейс для всех поддерживаемых алгоритмов. Класс Context использует этот интерфейс для вызова конкретного алгоритма, определенного в классе ConcreteStrategy.

- ConcreteStrategy (SimpleCompositor, TeXCompositor, ArrayCompositor) — конкретная стратегия:

- реализует алгоритм, используя интерфейс, объявленный в классе Strategy.

- Context (Composition) — контекст:

- настраивается объектом класса ConcreteStrategy.

- хранит ссылку на объект класса Strategy.

- может определять интерфейс, позволяющий объекту Strategy обращаться к данным контекста.

**Отношения**

- Классы Strategy и Context взаимодействуют для реализации выбранного алгоритма. Контекст может передать стратегии все необходимые данные в момент вызова алгоритма. Вместо этого контекст может позволить обращаться к своим операциям в нужные моменты, передавая ссылку на самого себя операциям класса Strategy.

- Контекст переадресует запросы своих клиентов объекту-стратегии. Клиент обычно создает объект ConcreteStrategy и передает его контексту, после чего взаимодействует исключительно с контекстом. Обычно у клиента есть несколько классов ConcreteStrategy, из которых он может выбирать.

**Результаты**

Основные достоинства и недостатки паттерна стратегия:

- Семейство родственных алгоритмов. Иерархия классов Strategy определяет семейство алгоритмов или вариантов поведения, которые можно повторно использовать в разных контекстах. Наследование позволяет выделить общую для всех алгоритмов функциональность.

- Альтернатива порождению подклассов. Наследование поддерживает многообразие алгоритмов или поведений. Можно напрямую породить от Context подклассы с различными поведениями. Но при этом поведение "зашивается" в класс Context, что затрудняет понимание, сопровождение и расширение контекста. Кроме того, заменить алгоритм динамически уже не удастся. Разделение алгоритма в отдельный класс Strategy позволяет изменять его независимо от контекста.

- Стратегии позволяют избежать условных конструкций. Паттерн стратегия позволяет избежать условных операторов при выборе нужного поведения. Когда различные поведения объединены в один класс, трудно выбрать нужное без использования условных операторов. Инкапсуляция каждого поведения в отдельный класс Strategy решает эту проблему.

**Пример использования:**

Без паттерна стратегия код для разбиения текста на строки мог бы выглядеть следующим образом:

```cpp

void Composition::Repair() {

switch (\_breakingStrategy) {

case SimpleStrategy:

ComposeWithSimpleCompositor();

break;

case TeXStrategy:

ComposeWithTeXCompositor();

break;

// ...

}

//

}

```

Паттерн стратегия позволяет обойтись без конструкции выбора за счет делегирования задачи разбиения на строки объекту Strategy:

```cpp

void Composition::Repair() {

\_compositor->Compose();

//

}

```

Если код содержит много условных операторов, то это часто является сигналом для применения паттерна стратегия.

- Выбор реализации. Стратегии могут предоставлять разные реализации одного и того же поведения. Клиент может выбирать подходящую стратегию в зависимости от своих требований к быстродействию и памяти.

- Клиенты должны знать о различных стратегиях. Недостаток паттерна в том, что клиент должен понимать различия между разными стратегиями для выбора подходящей. Поэтому, вероятно, придется раскрывать клиенту некоторые особенности реализации. Поэтому стоит применять этот паттерн только тогда, когда различия в поведении важны для клиента.

- Затраты на передачу информации между стратегией и контекстом. Интерфейс Strategy используется всеми подклассами ConcreteStrategy, независимо от сложности их реализации. Некоторые стратегии могут не использовать всю передаваемую информацию, особенно простые. Это может привести к созданию и инициализации ненужных параметров. Если возникнет проблема, то придется установить более тесную связь между классами Strategy и Context.

- Увеличение числа объектов. Применение стратегий увеличивает число объектов в приложении. Иногда эти издержки можно сократить, реализовав стратегии в виде объектов без состояния, которые могут использоваться несколькими контекстами. Остаточное состояние хранится в самом контексте и передается при каждом обращении к объекту-стратегии. Совместно используемые стратегии не должны сохранять состояние между вызовами. В паттерне Приспособленец (Flyweight) этот подход обсуждается более подробно.