**Вопрос 4. Рефакторинг и оптимизация программного кода**

## 1. Изменение структуры программного кода

**Рефакторинг или реорганизация кода** — процесс изменения внутренней структуры программного продукта, не затрагивающий её внешнего поведения и имеющий целью облегчение понимания программного кода и, пусть и не всегда, оптимизацию производительности.

Рефакторинг — это переработка исходного кода программы, чтобы он стал более простым и понятным. Рефакторинг не меняет поведение программы, не исправляет ошибки и не добавляет новую функциональность. Он делает код более понятным и удобочитаемым.

**Например**, *вот фрагмент на Python, создающий список из строки:*

```
list = []
for char in 'abcdef':
 if char != 'c':
  list.append(char * 2)
print(list) # ['aa','bb','dd','ee','ff']
```

*При рефакторинге его можно упростить, применив конструктор списков:*

```
list = [char * 2 for char in 'abcdef' if char != 'i']
print (list)
```

*Результат работы программы не изменился, но код стал проще, компактнее и понятнее.*

Последовательность таких небольших изменений может сильно улучшить качество проекта.

В основе рефакторинга лежит последовательность небольших преобразований программного кода, сохраняющих его поведение. Так как каждое преобразование по объёму незначительно, то программисту легче проследить за его правильностью, а вся последовательность этих изменений может привести к существенной перестройке программы и улучшению её согласованности, четкости и простоты понимания её кода другими разработчиками.

Обычно рефакторят код под контролем прохождения автоматизированных тестов, без этого достаточно сложно убедиться, что внесённые изменения не являются деструктивными и поведение программы осталось прежним.

**Без рефакторинга не обходится ни один действительно сложный и долгоживущий проект.** Обойтись без рефакторинга можно лишь в том случае, если разрабатывается что-то, что будет использовано лишь однажды, а потом просто выброшено. Во всех остальных случаях рефакторинг необходим.

### Применение рефакторинга

В самом простом случае, рефакторинг осуществляется в процессе написания кода. Разработчик реализует функционал, добивается работоспособности, а затем проводит оптимизацию и рефакторинг написанного кода. К сожалению, достаточно трудно написать сложный программный компонент идеально с первого раза: полное понимание взаимосвязей, логики и вариантов реализации, как правило, приходит в процессе разработки.

Тем не менее, использование рефакторинга только в процессе разработки отдельных компонентов не достаточно. Если разрабатываемый компонент не изолирован, а взаимодействует с другими, то обычно есть необходимость в рефакторинге программных интерфейсов, через которые это самое взаимодействие реализуется.

В рамках всей программной системы перед рефакторингом стоит еще задача унификации именования функций и переменных, форматирования и достижения соблюдения стандартов кодирования.

Стройный, хорошо структурированный код легко читается и быстро дорабатывается. Но редко удаётся сразу сделать его таким. Разработчики спешат, в процессе могут меняться требования к задаче, тестировщики находят баги, которые нужно быстро исправить, или возникают срочные доработки, и их приходится делать второпях.

В результате даже изначально хорошо структурированный исходник становится беспорядочным и непонятным. Программисты знают, как легко завязнуть в этом хаосе. Причём неважно, чужой это код или собственный.

Чтобы решить все эти проблемы, делается рефакторинг программы. В новом проекте он нужен, чтобы:

- сохранить архитектуру проекта, не допустить потери структурированности;
- упростить будущую жизнь разработчиков, сделать код понятным и прозрачным для всех членов команды;
- ускорить разработку и поиск ошибок.

**Рефакторинг очень существенно влияет на сопровождаемость проекта**

Любой проект без регулярного рефакторинга за несколько лет (или даже месяцев) становится трудным для понимания. Процессы изменений замедляются и становятся дороже, а иногда такие проекты доходят до состояния «проще переписать тут всё с нуля, чем разбираться». Таким образом затраты на рефакторинг окупаются за счёт того, что изменения вносить становится проще и процесс модернизации обходится значительно дешевле.

Но любое приложение со временем устаревает: язык программирования совершенствуется, появляются новые функции, библиотеки, операторы, делающие код проще и понятнее. То, что год назад требовало пятидесяти строк, сегодня может решаться всего одной. Поэтому даже идеальная когда-то программа со временем требует нового рефакторинга, обновляющего устаревшие участки кода.

**Наиболее частые причины для рефакторинга:**

- дублирование кода
- длинные методы
- объёмные классы
- длинные списки параметров
- избыточные временные переменные
- классы данных
- несгруппированные данные
- несоблюдение стандартов кодирования

**Когда нужно срочно улучшать код**

Признаки, показывающие, что назрела необходимость в рефакторинге:

- Программа работает, но даже небольшие доработки сильно затягиваются из-за того, что каждый раз приходится долго разбираться в коде.
- Разработчик постоянно не может точно сказать, сколько времени ему нужно на выполнение задачи, потому что «там надо вначале разбираться».

- Одинаковые изменения приходится вносить в разные места текста программы.

Такой код нужно срочно рефакторить, иначе он будет тормозить реализацию проекта и затруднять внесение правок.

Вообще рефакторинг нужно проводить постоянно. Делайте его каждый раз, после того как поменяли программу и убедились, что всё работает. Например, если добавили или изменили какую-то функцию, метод, класс или объявили новую переменную.

**Как делают рефакторинг**

Рефакторинг — это маленькие последовательные улучшения кода. Чистить можно всё, но в первую очередь найдите эти проблемы:

1. **Мёртвый код.** Переменная, параметр, метод или класс больше не используются: требования к программе изменились, но код не почистили. Мёртвый код может встретиться и в сложной условной конструкции, где какая-то ветка никогда не исполняется из-за ошибки или изменения требований. Такие элементы или участки текста нужно удалить.
2. **Дублирование.** Один и тот же код выполняет одно и то же действие в нескольких местах программы. Вынесите эту часть в отдельную функцию.
3. **Имена переменных, функций или классов не передают их назначение.** Имена должны сообщать, почему элемент кода существует, что он делает и как используется. Если видите, что намерения программиста непонятны без комментария, — рефакторьте.

**Примеры корректных имен:** *totalScore* — переменная, означающая итоговый счёт в игре, *maxWeight* — максимальный вес. Для функций и методов лучше использовать глаголы, например: *saveScore ()* — сохранить счет, *setSize ()* — задать размер, *getSpeed ()* — получить скорость.

4. **Слишком длинные функции и методы.** Оптимальный размер этих элементов — 2-3 десятка строк. Если получается больше, разделите функцию на несколько маленьких и добавьте одну общую. Пусть маленькие выполняют по одной операции, а общая функция их вызывает.
5. **Слишком длинные классы.** То же самое. Оптимальная длина класса — 20–30 строк. Разбейте длинный класс на несколько маленьких и включите их объекты в один общий класс.
6. **Слишком длинный список параметров функции или метода.** Они только запутывают, а не помогают. Если все эти параметры действительно нужны, вынесите их в отдельную структуру или класс с понятным именем, а в функцию передайте ссылку на него.
7. **Много комментариев.** Плохой код часто прикрывается обильными комментариями. Если почувствовали желание пояснить какой-то участок кода, попробуйте сначала его переписать, чтобы и так стал понятным. Бесполезные комментарии загромождают программу, а устаревшие и неактуальные вводят в заблуждение.

После каждой правки посмотрите на соседние участки кода: возможно, их тоже стоит поправить и сделать понятнее. И на те участки кода, которые давно не редактировались, — они уже могли стать некорректными.

После каждого изменения программу надо тестировать, поэтому перед началом рефакторинга подготовьте комплект тестов: модульных, функциональных или интеграционных. Изменения при рефакторинге вносятся небольшие, так что ошибки обычно легко найти и исправить.

Код чистят и на этапе тестирования, когда всё уже готово и проверяется работоспособность программы. Тут разработчик выполняет требования тестировщиков и одновременно проводит рефакторинг.

Как правило, руководители проектов понимают важность рефакторинга и делают его элементом разработки. Особое место он занимает в экстремальном программировании, когда программисты попеременно то пишут код и разрабатывают тесты, то проводят рефакторинг написанного.

**В чём опасности рефакторинга**

Мы всё-таки меняем рабочий код. Тут можно не только всё упростить, но и сильно напортачить. Небрежный рефакторинг может отбросить выполнение проекта на дни и недели.

Не страдайте перфекционизмом! Если вы поправили какой-то кусочек кода, не надо перетряхивать всю программу, разыскивая, что ещё можно улучшить. Стремление к совершенству вечно, но лучше обойтись без фанатизма.

Опасно делать рефакторинг не постоянно, а от случая к случаю. Соблазн сильно улучшить код становится невыносимым. Вы всё глубже закапываетесь в программу и копаете себе яму, в которой легко увязнуть. **Рефакторьте постоянно и по чуть-чуть.**

Иногда бывают злоупотребления: рефакторинг может стать способом саботажа, отговоркой, с помощью которой откладываются важные релизы и внедрение новых фич.

Но всё равно нельзя пренебрегать усовершенствованием кода, потому что это лучший способ ускорить работу в будущем.

## 2. Модификация кода

**Оптимизация программы –** это её модификация с целью повышения эффективности работы.

- Для оптимизации требуется найти критическую часть кода, которая является основным потребителем необходимого ресурса.
- Для поиска узких мест используются специальные программы — профайлеры.

**Зачем же нужна оптимизация и откуда она взялась?**

С первых дней развития эры вычислительной техники возник вопрос экономии места и увеличения производительности программ. Программистам приходилось создавать сложные дееспособные программы, которые смогли бы работать при очень низком быстродействии процессоров, использовать считанные килобайты оперативной памяти и места на диске. Поэтому все разработчики ПО были заинтересованы в максимальном быстродействии при минимальном размере кода.

Сегодня эти мощности вызывают улыбку. Но традиции оптимизации кода сохранились. Как известно, сколько ни наращивай размер диска и объем ОЗУ, все равно будет мало. Потому написанные «неряшливо» приложения, медленные и ресурсоемкие, проигрывают конкурентную борьбу аналогам, даже если они красивы и удобны.

Особо жесткие требования касаются драйверов и системных утилит. Они должны работать быстро, корректно и максимально экономить ресурсы компьютера. То есть взаимодействие процессора с периферией должно происходить без лишних затрат времени, с высокой скоростью передачи данных между устройствами. И сейчас мы решили немного разобраться, какие бывают способы оптимизировать программный код, в чем их плюсы и минусы.

**Основные принципы оптимизации**

Оптимизация стоит на трех «китах» — естественность, производительность, затраченное время. Давайте разберемся подробнее, что они означают.

1. **Естественность.** Код должен быть аккуратным, модульным и легко читабельным. Каждый модуль должен естественно встраиваться в программу. Код должен легко поддаваться редактированию, интегрированию или удалению отдельных функций или возможности без необходимости вносить серьезные изменения в другие части программы.
2. **Производительность**. В результате оптимизации вы должны получить прирост производительности программного продукта. Как правило, удачно оптимизированная программа увеличивает быстродействие минимум на 20-30% в сравнение с исходным вариантом.
3. **Время**. Оптимизация и последующая отладка должны занимать небольшой период времени. Оптимальными считаются сроки, не превышающие 10 – 15 % времени, затраченного на написание самого программного продукта. Иначе это будет нерентабельно.

Полезный совет. Перед началом оптимизации программного кода не забудьте сохранить копию исходного кода. Тогда в случае ошибки при внесении изменений всегда можно будет откатится до рабочей версии.

**Стоит ли применять Ассемблер**

Многие разработчики искренне считают, что критические секции (некоторые называют их «узкими» местами программы) кода удобнее писать на ассемблере, так как он обеспечивает самый быстрый доступ к данным и устройствам.

Многочисленные сравнения результатов оптимизации кода на языке высокого уровня и применения ассемблерных вставок показывают, что в первом случае после компиляции программа будет работать чуть медленнее, чем при использовании ассемблера. Обычно эти цифры измеряются от 2% до 7%. Максимум – разница составляет 20%. Стоит ли ради получения столь малого эффекта тратить время и силы на написание ассемблерной версии? На наш взгляд лучше уделить больше внимания работе с кодом, который написан на ЯП высокого уровня, и оптимизировать работу алгоритма.

**Как правильно оптимизировать?**

Теперь давайте разберемся, как проводится оптимизация, и разберемся, с чего начинать, чего лучше избегать, и когда без ассемблера не обойтись.

1. Начало оптимизации

Первое, что необходимо сделать, это выявить «узкие места» программы. Нет смысла трогать тот кусок программы, где и без вас все работает прекрасно. Здесь вы вряд ли что-то выиграете при оптимизации. В первую очередь, стоит обратить внимание на блоки кода, которые регулярно или часто повторяются в процессе работы – циклы и подпрограммы.

**Пример**: *Если оптимизировать работу цикла хотя бы на 2% за одну итерацию, а число его повторов будет 1000 раз, в итоге мы получаем: 2% × 1000 = 2000%, вполне ощутимый результат при работе кода.*

2. Участки кода, которые не оптимизируются

Не стоит трогать единичные операнды, поскольку работают они крайне редко и толку в их модификации нет никакого. Они отработают один раз, и больше к этому коду обращений не будет. Но при условии, что при внесении изменений вы добьетесь увеличения производительности более чем на 10%, это не лишено смысла. Здесь все зависит от вашего здравого смысла и опыта.

Также вы не сумеете добиться достойных результатов в случае обращения к внешним устройствам и другим программным системам. До и после таких фрагментов можно что-то ускорить. Но там, где задержка может возникать по причине взаимодействия с внешними данными, лучше предусмотрите заглушку типа «Подождите, операция может занять несколько минут».

**Еще раз об ассемблере**

Помните, что использовать ЯП низкого уровня нужно только там, где это действительно необходимо. Нет никакой причины вставлять его везде, это никак не повлияет на производительность. Впрочем, если вы – асс ассемблера и можете писать на нем также быстро, как и на удобном языке высокого уровня, можете пользоваться им активно. Правда, тогда возникает другой нюанс – вы усложняете читабельность кода для программистов, которые будут заниматься проектом после вас.

**Оптимизировать или нет?**

Если вы не уверены, что сможете ускорить работу программы, при этом тестирование не выявляет никаких критичных проблем, оставьте все как есть. Помните старую мудрость программистов: работает – не мешай.

Иначе вы можете потратить лишнее время на работу с кодом, а в результате сделаете даже хуже, программа начнет работать медленно, да и от багов никто не застрахован.

Заниматься оптимизацией следует только тогда, когда на программу поступают жалобы пользователей либо на этапе тестирования находятся проблемные участки, на которых программа «подвисает» и тормозит работу устройства. В таких случаях производится отладка, а для уже выпущенных в серию продуктов выпускают новые версии или, так называемые, «заплатки» (patch).

Оптимизация кода не слишком отличается от обычного исправления багов. Более того, с их устранения и начинается работа по оптимизации программы.

Первым делом нужно проверить код на наличие устаревших или вообще ненужных фрагментов. Таких исполняемых модулей или веток в большой программе находится обычно много. Что-то написали, но оказалось, что функционал не нужен, и его просто забыли удалить. Другие части оказались не нужны в результате очередного обновления. Все они занимают место. А некоторые продолжают исполняться, хоть в этом нет никакого смысла. И, таким образом, замедляют работу системы.

Пишите аккуратный код. Не забывайте о комментариях. Так вы поможете и себе, и другим разработчикам, понять, что в программе нужно, а что – уже не актуально. Эти общие советы помогают и при отладке, и при поиске багов. В общем, не будьте неряшливым «говнокодером», и ваши программы будут работать быстро и без проблем.

Второй этап поиска проблемных мест также простой. Разберитесь, когда приложение работает медленнее всего, в какие моменты оно заметно подвисает. Изучите код на предмет ошибок или излишне сложных запутанных решений. Попробуйте написать проще.

Если все равно что-то не работает или «тормозит», придется использовать профилировщики отладочного вывода, в том числе, с учетом записи логов всех запросов SQL, если программа работает с базами данных. В случае поиска вслепую вы потратите много времени и не факт, что сможете добиться положительных результатов.

**Рассмотрим самые популярные методы оптимизации программ. Некоторые из них возможно вызовут у вас недоумение, но поверьте, они работают.**

**Настройка окружения**

Используемая вами SQL или другая система управления базами данных могут быть неверно настроены. Настройки по умолчанию далеко не идеальны. Возможно, какие-то дополнительные проверки как раз и замедляют процесс.

Иногда удается заметно ускорить работу программы, изменив ключевые настройки виртуальной машины Java. Кстати, это поможет сделать быстрее работу не только тестируемого приложения, но и всей системы.

Также обратите внимание на саму операционную систему и мощность «железа». Может быть они вообще не предназначены для работы программного продукта, который вы пытаетесь запустить и ускорить? А, может, устарели и потому «тормозят»?

Все это не относится напрямую к оптимизации программы, но проверить нужно до начала работы с кодом. Просто потому, что такие «досадные мелочи» нередко оказываются ключевой проблемой, а код – вообще не причем. И не стоит снисходительно улыбаться. Проверять окружение забывают даже опытные разработчики.

**Избавляемся от ненужного функционала**

Для увеличения скорости работы приложения можно использовать подход сокращения ненужного кода. Часто бывает так, что программа автоматически решает маловажные или уже не актуальные задачи. Например, заказчик, описывая задачу программисту, попросил о каких-то возможностях, а потом передумал. Или вышел новый релиз программы, где часть функций выделили в отдельный модуль, а старый код просто забыли удалить.

В итоге мы имеем лишний функционал, который будет «тормозить» быстродействие. Со временем такой код обрастет совершенно ненужными «костылями» и «подпорками», что не лучшим образом влияет на производительность. В таком случае мы рекомендуем просто переписать модуль «с нуля», выбросив все старое, как ненужных хлам.

**Меморизация** (от англ. Memorization) означает запоминание. Фактически это простое сохранение результата выполнения определенной функции, которое поможет избежать ее повторного выполнения. Применяя меморизацию, вы сможете повысить производительность программы.

Работает это очень просто. Перед тем, как функция будет выполняться, проверяется условие – исполнялась ли она ранее. По итогам можно получить два варианта:

- функция вызвана в первый раз, тогда она выполняется, а результат сохраняется;
- модуль уже работал, можно использовать сохраненный результат.

Иногда говорят о табулировании, это синоним мемоизации, который используется во многих языках программирования.

**Кеширование** – метод временного хранения данных в памяти устройства пользователя. Получить доступ к такой информации можно гораздо быстрее, чем каждый раз обращаться к серверу или базам данных. С помощью кэширования значительно ускоряют работу с сайтами, онлайн-системами и т.д.

Вся необходимая информация в данном случае храниться на носителях с быстрым доступом. Это может быть выделенная часть диска или оперативная память. Программа в процессе работы использует кэш по мере необходимости, и обращается к основному хранилищу данных только если не находит их в кэше.

**Распараллеливание программ**

Это способ адаптации алгоритмов, которые были реализованы, как программы для компьютерных систем с параллельной архитектурой. Как правило, это относится к многопроцессорным системам.

Подробно описывать метод мы здесь не будем, так как это займет слишком много места. Кратко можно сказать так: разные вычисления одной программы выполняются одновременно в параллельных потоках. Такой подход позволяют далеко не все языки, а потому тут нередко используют внешние команды системы или ассемблер.

**«Ленивые» вычисления**

Ленивые (Lazy evaluation) или отложенные вычисления – стратегия, которую применяют в некоторых системах счисления. Суть метода заключается в том, что все расчеты откладываются до тех пор, пока не будет затребован их результат.

Такая стратегия позволит существенно снизить общий объем производимых вычислений, так как ненужные операции попросту не будут выполняться. Чтобы метод начал работать, нужно описать зависимости функций (операндов) друг от друга, что поможет отслеживать работу. В итоге вы получите код, который будет выполняться только в том случае, когда он действительно нужен.

**Метод приближения**

Приближение или аппроксимаация (от лат. proxima — ближайшая или приближе́ние) – метод замены строгого алгоритма на наиболее подходящие приближенные значения, что влечет за собой определенную потерю точности вычислений. Снижение точности экономит память и повышает скорость. Для того чтобы не использовать длинную арифметику, можно воспользоваться обычными float'ами. Но пользоваться таким методом нужно крайне осторожно, не всегда снижение точности допустимо.

**Использование сторонних языков**

Иногда написанная программа может медленно работать из-за того, что много времени занимает проверка описанных типов, что занимает дополнительное время. Чтобы избежать этого эффекта, можно применять фрагменты кода или модули, написанные на других языках. Но делать это нужно крайне осторожно. Все эти «лишние» проверки защищают вас от багов и «дыр» в безопасности, связанных, в том числе, с буферизацией. Потому хорошо подумайте, действительно ли экономия времени столь существенна? И если придете к выводу, что здесь это – лучшее решение, обязательно проведите особо внимательное тестирование.

Кроме того, если начать использовать в коде фрагменты других языков, это может вызвать эффект «зоопарка», что сильно снижает читабельность программы. Также имейте в виду, что метод может попросту не сработать или даже критически навредить всей программе.

## 3. Отличия рефакторинга и оптимизации производительности

Стоит разделять эти понятия. Оптимизация кода, нацеленная на обеспечение быстродействия, иногда приводит к снижению понятности кода: часто эффективный код — это код, написанный не так, как понятно человеку, а так, как понятно компьютеру.

Но рефакторинг сам по себе нередко повышает производительность, так как выявляются и удаляются лишние конструкции, от которых результат работы не зависит, но на время выполнения программы такие конструкции могут существенно влиять; иногда меняется и последовательность выполнения программного кода, что тоже может позитивно повлиять на производительность.

Рефакторинг — не оптимизация, хотя и может быть с нею связан. Часто его проводят одновременно с оптимизацией, поэтому понятия кажутся синонимами. Но у этих процессов разные цели.

Цель оптимизации — улучшение производительности программы, а рефакторинга — улучшение понятности кода. После оптимизации исходный код может стать сложнее для понимания. После рефакторинга программа может начать работать быстрее, но главное — её код становится проще и понятнее.

Литература:

1. Рефакторинг — это неизбежный процесс / https://web-creator.ru/articles/refactoring
2. Демидова М. Что такое рефакторинг кода и зачем он нужен / https://skillbox.ru/media/code/chto_takoe_refaktoring_koda_i_zachem_on_nuzhen/
3. Оптимизация программного кода / https://techrocks.ru/2019/01/25/code-optimization-tips/
https://studfile.net/preview/4599549/page:38/ - еще про опт-ю, если не хватит