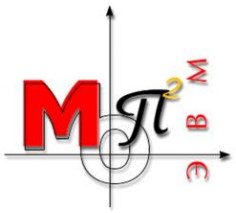
МИНОБРНАУКИ РОССИИ
Федеральное государственное автономное образовательное учреждения высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт компьютерных технологий и информационной безопасности
Кафедра математического обеспечения и применения ЭВМ



«Оптимизация и рефакторинг Java приложений в тг боте на Python»

ОТЧЁТ

по дисциплине

«Прикладное решение на языке Python»

Выполнил:
Студент группы
КТмо2-16

_____ Руденко К. Д.
подпись

Проверил:
Доцент кафедры САиТ, к.т.н.

_____ Лапшин В. С.
подпись

Оценка

_____

«____» _____ 2025 г.

Таганрог, 2025 г

Оглавление

Постановки задачи

Целью работы было изучение методов оптимизации и рефакторинга Java-приложений.

Задачи:

1) Изучение принципов оптимизации и рефакторинга ПО.

2) Определение различных граней приложения, которым нужно провести оптимизацию и рефакторинг.

3) Проанализировать код программы приложения с использованием различных инструментов статического анализа.

4) Найти проблемные зоны в производительности ПО и структуре приложения.

5) Использовать нужные техники оптимизации и рефакторинга для устранения найденых проблем.

6) Изменить приложение после проделанного тестирования.

7) Провести оценку эффективность проделанных мероприятий по оптимизации и рефакторингу.

8) Разработать приложение — телеграмм-бота со следующим функционалом: помощь в изучение основных методов рефакторинга с примерами и подробным объяснением и анализа кода java, через текст и через отправку файла.

9) Протестировать telegram бота.

Введение

В наше время современные интернет-технологии развиваются очень быстро. Для того чтобы они соответствовали требованиям, IT-специалисты способны использовать некоторые нетрадиционные методы создания функций архитектуры ПО. «Альтернативный» код становится все более популярным, что в результате приводит к тому, что структура ПО теряет четкость и понятность, а весь код становится все менее читаемым, поддерживаемым и легким к восприятию. Совсем небольшие изменения кода могут привести к непредсказуемым и различным ошибкам. Масштабная разработка ПО в таких суровых условиях может привести к катастрофическим и ужастным последствиям. Рефакторинг — это перестройка структуры всего кода в ПО с сохранением существующей функциональности без изменения логики, чтобы вновь добиться читаемости, структурированности и масштабируемости. Этот процесс улучшения кода позволяет повысить эффективность и уменьшить время разработки ПО и существенно оптимизировать производительность программы. Сам объем рефакторинга может быть различным — от отдельных связанных модулей до продукта целиком или даже отдельных функций. Если в проекте не подразумевается наличие ошибок в режиме реального времени, то необходимо сконцентрироваться на определенных аспектах, а не на мелких деталях в целом. Тем не менее, если имеется есть достаточное количество времени для разработки, и поддержки и анализа кода, то важно принимать во внимание учитывать каждую деталь, которую можно оптимизировать. В результате при оптимизации даже мелких частей в конечном итоге Накопление мелких оптимизаций определенно повысит эффективность работы приложения целикомкода.

1. Зачем нужна оптимизация и рефакторинг

Рефакторинг —— это процесс улучшения внутренней структуры ПО-программного кода без изменения его внешнего поведения. Это методическая и систематическая практика, направленная на повышение качества кода, его читаемости и сопровождаемости. Рефакторинг включает в себя устранение избыточности, упрощение сложных конструкций и улучшение архитектуры системы, что в конечном итоге способствует увеличению производительности, снижение затрат времени и ресурсов и облегчению внесения будущих изменений. Рефакторинг относиться к улучшение кода ПО, не меняя её работу и поведение.

Изменения в требованиях и устаревание технологий это факторы, которые негативно сказываются при использовании ПО. ПО создается для решения конкретных задач и при изменении требований его нужно обновлять. Не все изменения нужно внедрять из-за затрат и времени, но тем не менее, программное обеспечение должно адаптироваться к новым требованиям, чтобы оставаться востребованным. Прежде чем внедрять изменения, важно оценить их влияние на систему, включая возможные риски и выгоды. Обновление ПО может также включать улучшение его производительности, безопасности и удобства использования. Таким образом, своевременная адаптация к новым требованиям помогает сохранить конкурентоспособность и удовлетворить потребности пользователей.

Изначально программные продукты спроектированы с хорошей архитектурой. Но со временем и изменением требований нужно постоянно изменять функции и добавлять новые. При этом неизбежно возникают ошибки, которые требуют исправления. Чтобы внедрить изменения, приходится нарушать исходную архитектуру. Со временем в ней появляются «дыры». Ошибок становится все больше, поддерживать систему и добавлять новые функции становится сложнее. Архитектура теряет способность поддерживать новые функции и появляются сложности в их внедрении.

В итоге, затраты на разработку новых функций превысят затраты на создание нового ПО, и продукт становится невостребованным и умирает.В

листинге 1 показан пример, как уменьшить размер программы.

Листинг 1. fun_max

```
// Не обдумыный алгоритм нахождения максмимального числа
public boolean max(int x, int y) {
    if(x > y) {
        return true;
    } else if(x == y) {
        return false;
    } else {
        return false;
    }
}

// Самый короткий алгоритм
public boolean max(int x, int y) {
     return x > y;
}
```

Получился очень громоздкий код в первом случае - 6 строчек кода. Блок кода if-else не нужен. Такой код намного легче читать глазами, что уменьшает время анализа ПО.

1.1. Концепции рефакторинга и оптимизации

Рефакторинг и оптимизация помогают выявлять и решать различные проблемы, связанные с поддержкой и масштабируемостью.

Одной из часто требующих внимания проблем является наличие повторяющегося кода, который может привести к раздутым и трудноуправляемым кодом. Также слишком длинные и запутанные методы могут затруднить понимание и модификацию кода.

Более того, избыточная коммуникация между различными частями кода, проявляющаяся через сильно связанные классы и запутанные цепочки обращений, может еще больше осложнить ситуацию. Это может привести к запутанной сети зависимостей, усложняющей внесение даже небольших изменений без непредвиденных последствий.

Кроме того, устаревшие или неполные проектные решения могут сде-

лать архитектуру жесткой и нерасширяемой, подобно одноразовым структурам, которые не способны адаптироваться к изменяющимся требованиям. Отсутствие должной документации, такой как комментарии, объясняющие причины определенных решений или цель конкретных сегментов кода, только усугубляет путаницу и трудность поддержания в течение долгого времени.

Путем выявления и решения этих проблем через систематический рефакторинг разработчики могут оптимизировать код, сделав ее более модульной, поддерживаемой и масштабируемой. Это включает разбиение чрезмерно сложных методов на более мелкие и управляемые, уменьшение дублирования кода и разрыв тесно взаимосвязанных компонентов.

Более того, рефакторинг может также включать перестройку кода в соответствии с установленными шаблонами и принципами проектирования, обеспечивая его гибкость и адаптивность к будущим изменениям. Это может включать упрощение излишне запутанных коммуникационных путей, удаление ненужных слоев абстракции и стимулирование более четких и прямых взаимодействий между различными частями кода.

В конечном итоге, вложив время и усилия в рефакторинг, команды могут снизить риск деградации кода со временем, обеспечивая ее устойчивость и готовность к изменяющимся требованиям, хорошо рефакторизированный код лучше справляется с требованиями постоянного развития и поддержки.

1.2. Рефакторинг баз данных и интерфейса

Рефакторинг может привести к значительным изменениям в работе ПО. При рефакторинге базы данных необходимо отслеживать следующие моменты: тесно связанные структуры, миграция данных, производительность запросов, согласованность данных, изменение схемы базы данных, управление транзакциями, резервное копирование и восстановление, оптимизация индексов и обеспечение безопасности данных.

Следующий пример демонстрирует, как можно решить проблемы, связанные с тесно сопряженными структурами и сложностью миграции данных, с помощью рефакторинга базы данных. На рисунке 1 отображена база данных из одной таблицы в которой могут возникнут следующие проблемы:

1) Тесно связанные структуры — вся информация о заказах, доставке и оплате хранится в одной таблице.

2) Миграция данных — сложно вносить изменения в структуру таблицы без риска потери данных.

3) Сложность поддержки — изменения в бизнес-логике требуют изменений в одной громоздкой таблице.

order_id	customer_id	product_id	quantity	price	order_date	shipping_address	payment_info
1	101	5001	2	20.00	22.05.2024	Ленина 1	Мир Пэй
2	102	5002	1	15.00	23.05.2024	Пушкина 3	Сбер Пэй
3	101	5003	5	50.00	24.05.2024	Московская 4	Тинкофф Пэй

Рис. 1. таблица «order»

Рефакторинг предполагает разделение этой таблицы на несколько логически связанных таблиц. Поэтом новая структура данных будет выглядит как показано на рисунках 2 — 4.

order_id	customer_	order_date
1	101	22.05.2024
2	102	23.05.2024
3	101	24.05.2024

Рис. 2. таблица «orders»

item_id	order_id	product_id	quantity	price
1	1	5001	2	20.00
2	2	5002	1	15.00
3	3	5003	5	50.00

Рис. 3. таблица «order_items»

shipping_id	order_id	shipping_address
1	1	Ленина 1
2	2	Сбер Пэй
3	3	Ленина 1

Рис. 4. таблица «shipping_details»

payment_id	order_id	payment_info
1	1	Мир Пэй
2	2	Сбер Пэй
3	3	Мир Пэй

Рис. 5. таблица «payment_details»

Такой подход улучшает производительность и поддерживаемость системы.

При рефакторинге, который затрагивает изменения интерфейса могут возникать следующие проблемы: если что-то изменится неправильно, возникают сложности с поддержанием «опубликованного интерфейса»; необходимо поддерживать как старую, так и новую версии интерфейса, а также множество их вариантов; разные версии интерфейса должны работать корректно.

При необходимости изменить название функции, лучше оставить старую версию без изменений и вызывать её в новой функции с правильным названием. В «Java» старые методы можно пометить комментариями «устарело». Таким образом, разработчики заметят изменения.

Поддержание старого интерфейса может быть утомительным и ресурсозатратным. Придётся создавать и поддерживать дополнительные функции, по крайней мере, некоторое время. Они усложняют интерфейс и делают его менее удобным в использовании. Если переход от одного дизайна к другому кажется сложным через рефакторинг, тогда нужно уделить больше внимания начальному проектированию. Если есть простой способ для рефакторинга, лучше создать более простой дизайн.

1.3. Когда не следует проводить рефакторинг

Есть несколько ситуаций, когда проведение рефакторинга может быть нецелесообразным или даже вредным для проекта:

1) Когда нет времени. Если проект находится в критическом состоянии и требует быстрого завершения, проведение рефакторинга может быть нецелесообразным из-за риска задержки в сроках. В таких случаях приоритет должен быть уделен исправлению непосредственных проблем и завершению проекта вовремя.

2) Когда код находится в стабильном состоянии. Если код работает стабильно, без проблем и не требует изменений, то проведение рефакто-

ринга может быть излишним. В таких случаях лучше сконцентрироваться на других задачах, которые требуют больше внимания и улучшений.

3) Когда изменения могут нарушить внешний интерфейс. Если предстоящие изменения могут значительно изменить внешний интерфейс или API, то проведение рефакторинга может вызвать проблемы у клиентов, использующих этот интерфейс. В таких случаях лучше отложить рефакторинг до момента, когда изменения будут более предсказуемыми и меньше влияют на внешний мир.

4) Когда рефакторинг не решит основные проблемы. Иногда код может иметь серьезные архитектурные проблемы, которые не могут быть решены простым рефакторингом. В таких случаях необходимо провести более фундаментальные изменения в архитектуре проекта, что может потребовать более серьезных усилий и времени.

2. Подготовка к рефакторингу и оптимизации

Подготовка к рефакторингу и оптимизации включает в себя несколько этапов, которые позволяют эффективно и безопасно вносить изменения в существующий код:

• Понимание существующего кода. Важно полностью понять, как работает текущий код, его структуру и логику. Это включает в себя изучение функций, классов, зависимостей и алгоритмов, используемых в программе.

• Идентификация проблемных мест. Необходимо выявить участки кода, которые нуждаются в улучшении. Это могут быть участки с повторяющимся кодом, сложными конструкциями, низкой производительностью или неэффективными алгоритмами.

• Составление плана действий. На основе выявленных проблемных мест разрабатывается план поэтапного улучшения кода. При этом важно определить, какие конкретные изменения необходимо внести и в каком порядке их следует проводить.

• Создание тестового покрытия. Перед началом рефакторинга необходимо создать тесты, которые проверяют правильность работы кода. Это обеспечит безопасность при внесении изменений и поможет предотвратить появление новых ошибок.

• Постепенное внесение изменений. Рефакторинг следует проводить поэтапно, внедряя каждое изменение по мере готовности. Это позволяет контролировать процесс и минимизировать риск возникновения проблем.

• Проверка и анализ результатов. После завершения рефакторинга необходимо протестировать код и проанализировать его производительность. Это позволяет убедиться в том, что внесенные изменения действительно улучшили качество и эффективность программы.

2.1. Цель рефакторинга и оптимизации

Цель рефакторинга и оптимизации заключается в улучшении качества и эффективности программного кода. Это достигается путем перера-

ботки уже существующего кода с целью повышения его читаемости, понятности, модульности и поддерживаемости.

Рефакторинг направлен на устранение технического долга (ТД описывает будущие затраты времени и ресурсов, необходимые для исправления недостатков и ограничений, накопленных в программном коде из-за решений, принятых для ускорения разработки в краткосрочной перспективе), то есть на улучшение структуры и организации кода без изменения его функциональности. Оптимизация, в свою очередь, направлена на улучшение производительности программы путем оптимизации алгоритмов, устранения узких мест и улучшения времени выполнения или потребления ресурсов.

Объединение рефакторинга и оптимизации позволяет создавать более чистый, эффективный и легко поддерживаемый код, что в свою очередь способствует улучшению процесса разработки, уменьшению вероятности возникновения ошибок и повышению общего качества программного продукта.

Цель рефакторинга — поддерживать код в читаемом состоянии. Основная задача заключается в оптимизации дизайна, что привносит несколько выгод:

1) Улучшите понимание: поскольку программа в основном предназначена для людей, а не для машин, понятный и чистый код облегчает его анализ и поддержку.

2) Снижение затрат на модификацию: время обслуживания программы намного превышает время разработки. Хорошо структурированный код позволяет быстрее и безопаснее вносить изменения.

3) Увеличение скорости программирования: хороший дизайн является основой для быстрой разработки, поскольку понятный и модульный код облегчает процесс написания нового функционала.

2.2. Принципы рефакторинга и оптимизации

Принципы рефакторинга и оптимизации направлены на улучшение качества и эффективности программного кода.

• Сохранение функциональности — важно сохранять функциональность программы при проведении рефакторинга и оптимизации, чтобы избежать появления новых ошибок.

• Постепенность — рефакторинг и оптимизация должны проводиться постепенно и осторожно, чтобы минимизировать риск возникновения ошибок и негативных последствий.

• Систематичность — рефакторинг и оптимизация должны быть систематическими и продуманными действиями, основанными на анализе кода и его потребностей.

• Приоритетность — необходимо определить приоритетные области для рефакторинга и оптимизации, и начать с наиболее критически важных частей программы.

• Тестирование — после внесения изменений необходимо провести тестирование программы, чтобы убедиться в ее корректной работоспособности и отсутствии ошибок.

• Документирование — важно документировать внесенные изменения, чтобы облегчить понимание и поддержку кода другими разработчиками в будущем.

• Обучение и развитие — разработчики должны постоянно совершенствовать свои навыки рефакторинга и оптимизации, изучая новые методики и лучшие практики.

3. Техники рефакторинга

Введение в техники рефакторинга важно для понимания того, как можно систематически улучшать качество кода, делая его более понятным, поддерживаемым и масштабируемым. Существует множество техник рефакторинга, каждая из которых предназначена для решения специфи-

ческих задач по улучшению кода. В данной работе я рассмотрю наиболее основные и часто используемые техники, которые помогут сделать код более чистым, понятным и легким в сопровождении.

- Выделение класса (Extract Class)
- Выделение метода
- Передача всего объекта
- Отказ от наследования
- Ввод переменной
- Заменить числовые значения на константы
- Упрощение условных выражений
- Инкапсуляция полей
- Замена временной переменной запросом
- Объединить дублирующий код

3.1. Выделение класса

Выделение класса (Extract Class) — это рефакторинг, при котором часть функциональности одного класса переносится в новый класс. Этот процесс осуществляется для повышения читаемости, уменьшения размера класса и улучшения его понимания и поддержки. Выделение класса помогает соблюдать принцип единственной ответственности (Single Responsibility Principle), что делает код более модульным и лёгким в сопровождении. В приложении (листинг 2) показан пример выделения класса — класс выполняет слишком много функций поэтому, часть необходимо вынести в другой класс. На рисунке 6 можно увидеть наглядное представление выделения класса.

Шаги для выделения класса:

1) Определение функциональности. Найдите группу полей и методов в существующем классе, которые логически связаны и могут быть объединены в новый класс.

2) Создание нового класса: Создайте новый класс, который будет содер-

жать выделенную функциональность.

3) Перенос полей и методов. Переместите выбранные поля и методы из исходного класса в новый класс.

4) Адаптация исходного класса. Обновите исходный класс так, чтобы он использовал новый класс. Обычно это включает добавление объекта нового класса в исходный класс и изменение методов, которые обращаются к перенесённым полям и методам.

5) Обновление зависимостей. Обновите код, который использует исходный класс, чтобы он корректно взаимодействовал с новым классом.

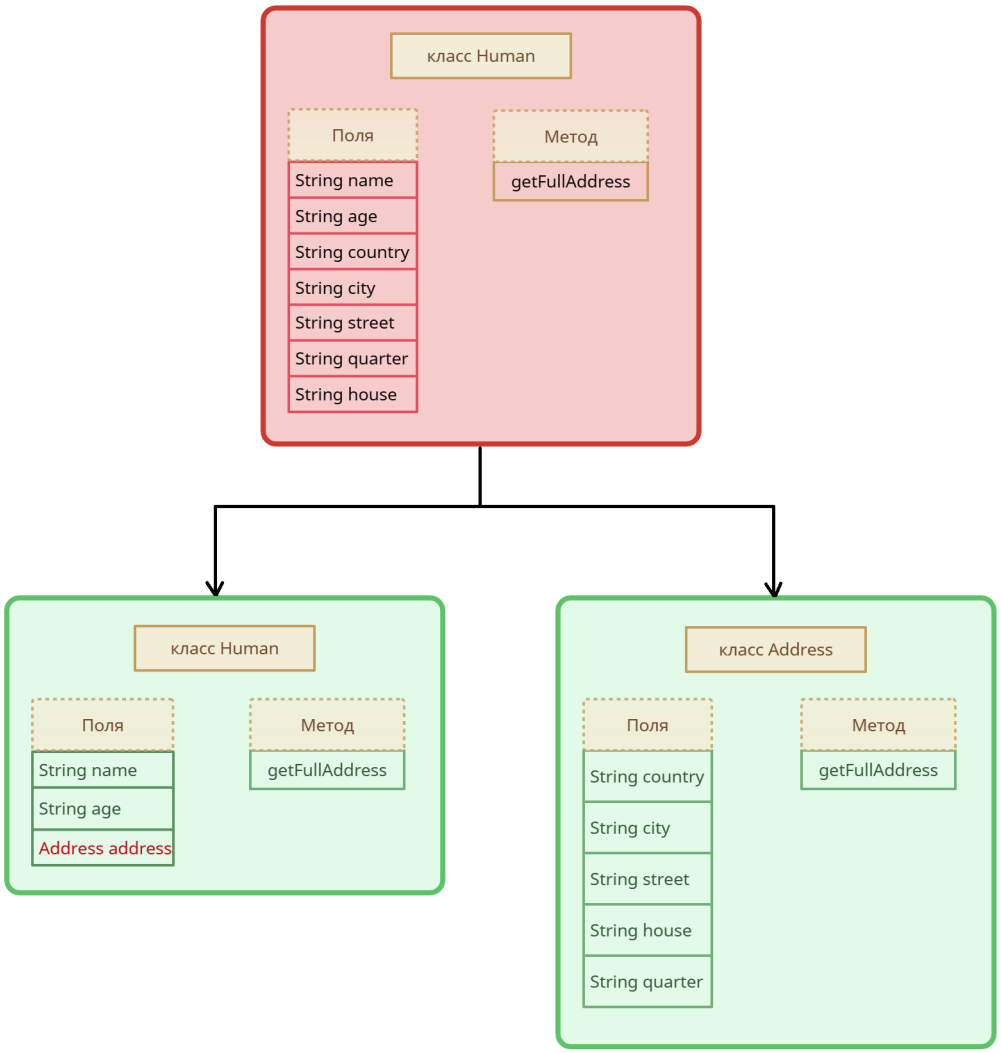


Рис. 6. Выделение класса

В классе до рефакторинга данные о адресе хранятся непосредственно в классе Human. Это приводит к нарушению принципа единственной ответственности (Single Responsibility Principle), так как класс Human должен отвечать только за свою собственную информацию о человеке, а не за информацию об адресе.

В классе после рефакторинга данные об адресе вынесены в отдельный класс Address. Это позволяет избежать дублирования информации и обеспечивает лучшую организацию кода. Каждый класс отвечает только за свою область ответственности: Human — за информацию о человеке, Address — за информацию об адресе. Такой подход соблюдает принцип единственной ответственности и делает код более читаемым, понятным и поддерживаемым.

Другим преимуществом рефакторинга является повышение гибкости и расширяемости кода. Если в будущем потребуется добавить или изменить информацию об адресе (например, добавить поля для почтового индекса или номера квартиры), это можно будет сделать без изменения класса Human. Просто потребуется внести соответствующие изменения в класс Address, что сократит вероятность возникновения ошибок и упростит процесс разработки.

3.2. Выделение метода

Выделение метода заключается в вынесении группы операций или функционала из основного метода в отдельный метод. Примером выделения метода может служить реализация метода alcQuadraticEq исходная схема показана на рисунке 7, который вычисляет корни квадратного уравнения. Весь код продемонстрирован в листинге 3.

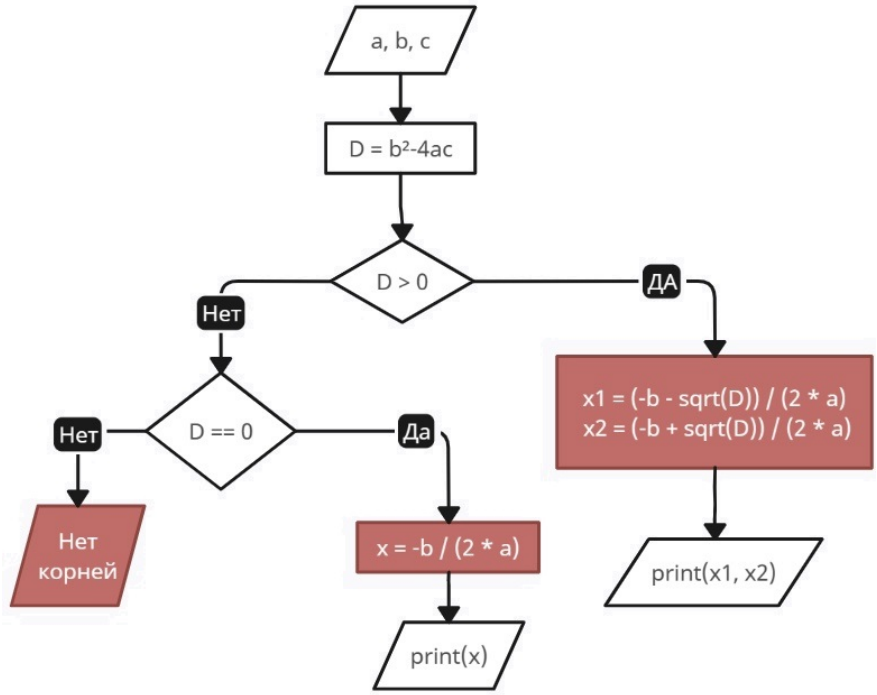


Рис. 7. Выделение метода до рефакторинга

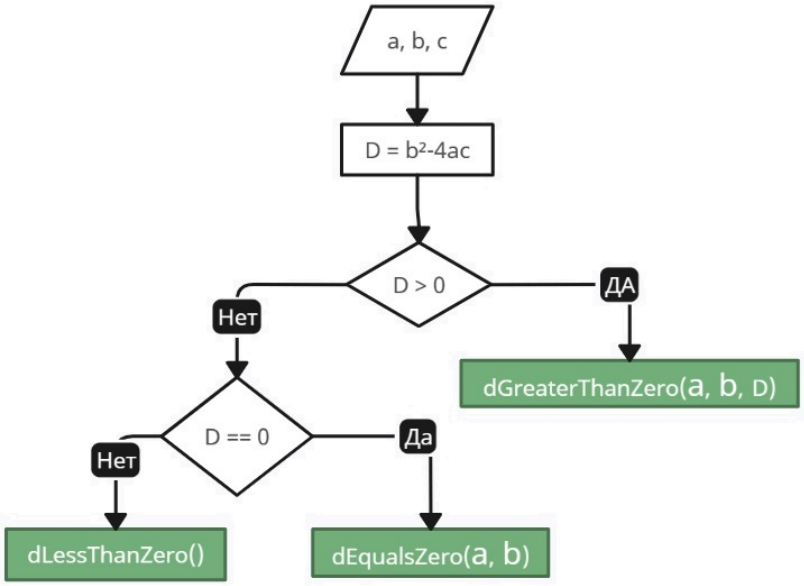


Рис. 8. Выделение метода

В исходной реализации этот метод содержит несколько операций, связанных с проверкой дискриминанта и вычислением корней. Однако, используя прием выделения метода, мы можем разбить этот метод на более

мелкие и понятные части, вынося каждый из вариантов вычисления корней в отдельный метод, это показано на рисунке 8.

Такой подход делает код более модульным и читаемым. Каждый выделенный метод отвечает только за свою часть логики, что упрощает его понимание и поддержку. Кроме того, такой подход делает код более гибким, поскольку позволяет легко добавлять новую логику или изменять существующую без необходимости внесения изменений в другие части программы.

3.3. Передача всего объекта

В процессе разработки программного обеспечения часто встречаются ситуации, когда методы класса требуют передачи множества параметров для выполнения своих функций. Это может привести к избыточности кода и уменьшению его читабельности и поддерживаемости. В исходном коде на рисунке 9, метод employeeMethod принимает объект Employee и извлекает из него годовую зарплату и сумму премий, после чего передает эти данные в метод getMonthlySalary, который рассчитывает ежемесячную зарплату. Полный код представлен в приложении (листинг 4).

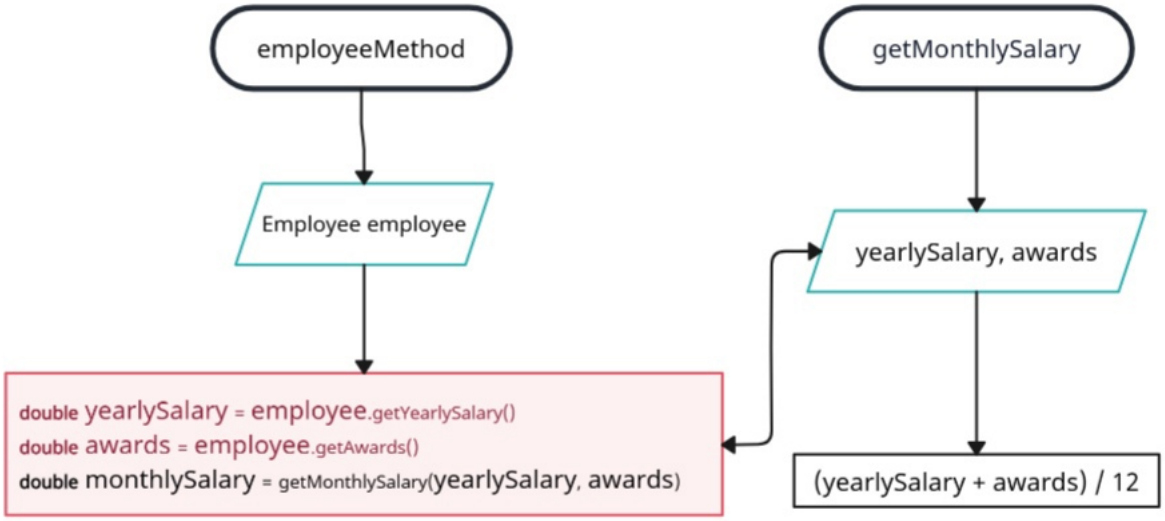


Рис. 9. Передача всего объекта до рефакторинга

В оптимизированной версии метода employeeMethod упрощается пе-

редача параметров, путем непосредственной передачи объекта Employee в метод getMonthlySalary. Такой подход позволяет сократить количество кода, улучшить его читаемость и уменьшить вероятность ошибок при передаче параметров, продемонстрирован на рисунке 10.

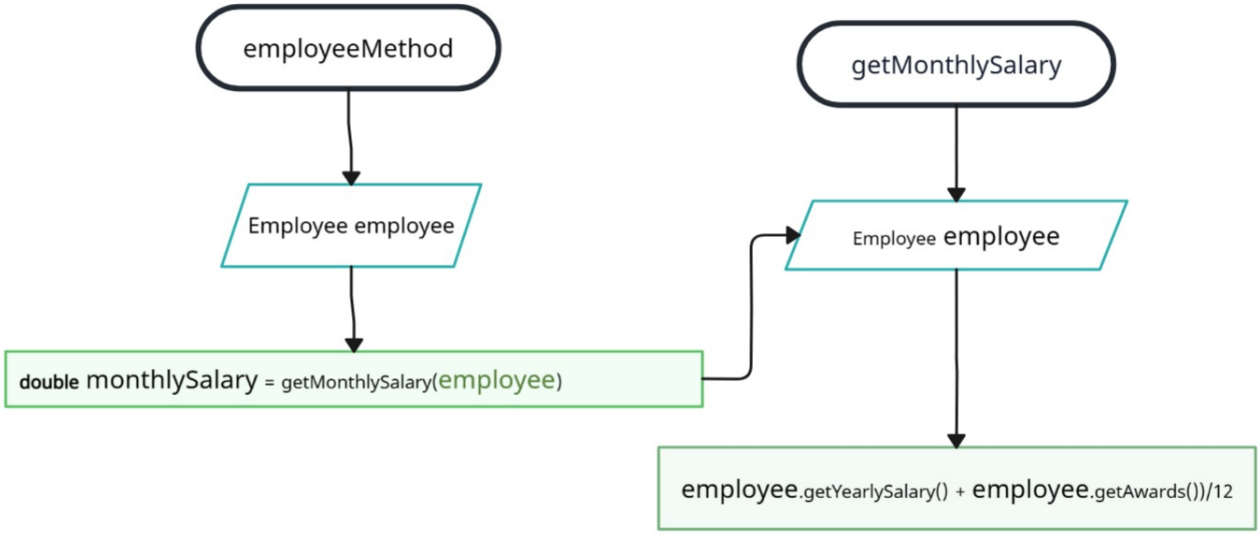


Рис. 10. Передача всего объекта

Описанная оптимизация представляет собой пример эффективной передачи данных внутри программы, где вместо множества отдельных параметров используется один объект, содержащий всю необходимую информацию. Это упрощает структуру кода, облегчает его понимание и поддержку, а также повышает гибкость системы в целом, поскольку изменения в структуре данных объекта Employee потребуют корректировки только внутри методов этого класса, не затрагивая интерфейс методов.

3.4. Отказ от наследования

Отказ от наследования — это принцип проектирования, который подразумевает, что подкласс не должен наследовать от родительского класса, если он использует только небольшую часть его функциональности или если связь между классами не является подходящей. Вместо этого предпочтительнее использовать композицию или делегацию для повторного использования кода. Правильная иерархия классов должна отражать отно-

шения «является/is-a», при которых подкласс является более специализированным типом своего суперкласса. Этот пример показан на рисунке 11.

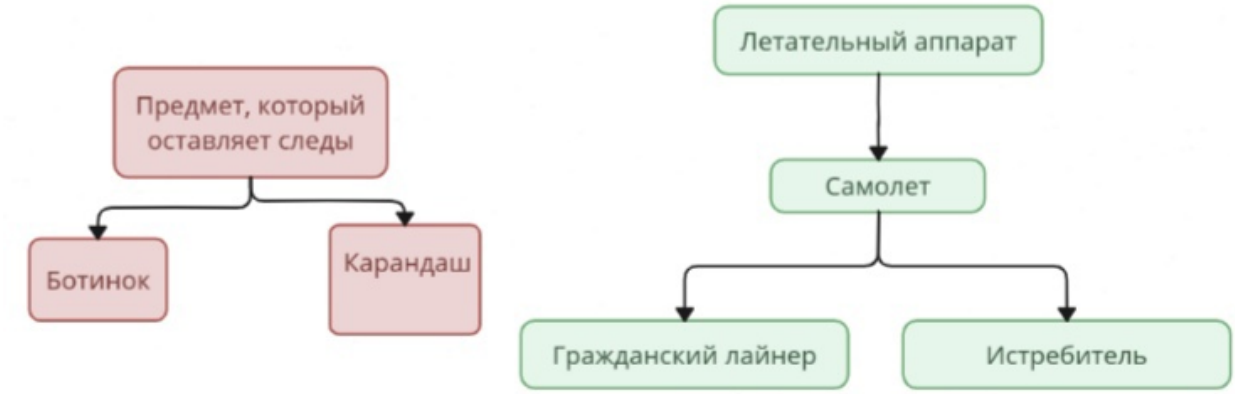


Рис. 11. Отказ от наследования

Слева на рисунке 11 показана неправильная иерархия классов, потому что класс «Предмет, который оставляет следы» абстрагирует понятие предмета, который оставляет следы. Он может иметь методы для оставки следов, например, leaveTrace() и предметы «Ботинок» и «Карандаш» не являются специализированными типами. А иерархия класов, которая представлена справа на картинке правильная потому что объекты «Гражданский лайнер» и «Истребитель» могут быть подклассами «Самолета», потому, что они представляют собой конкретные типы «летательных аппаратов». Каждый из них обладает характеристиками летательного аппарата и способностью летать, что является общим для всех летательных аппаратов.

4. Телеграмм бот

При разработке телеграмм бота был использован язык программирования Python с библиотеками: subprocess, Translator, python-telegram-bot. Для установки перечисленных библиотек использовалась команда:

pip install имя библиотеки

Следующим этапом для начала разработки необходимо получить токен. Для этого были выполнены следующие шаги:

1) Команда /newbot в группе BotFather.

2) Название для бота «Руденко К.Д ЮФУ ИКТИБ».

3) Создание короткой ссылке refactoring_java_bot.

На рисунке 12 показана история создания бота с описанными выше шагами.

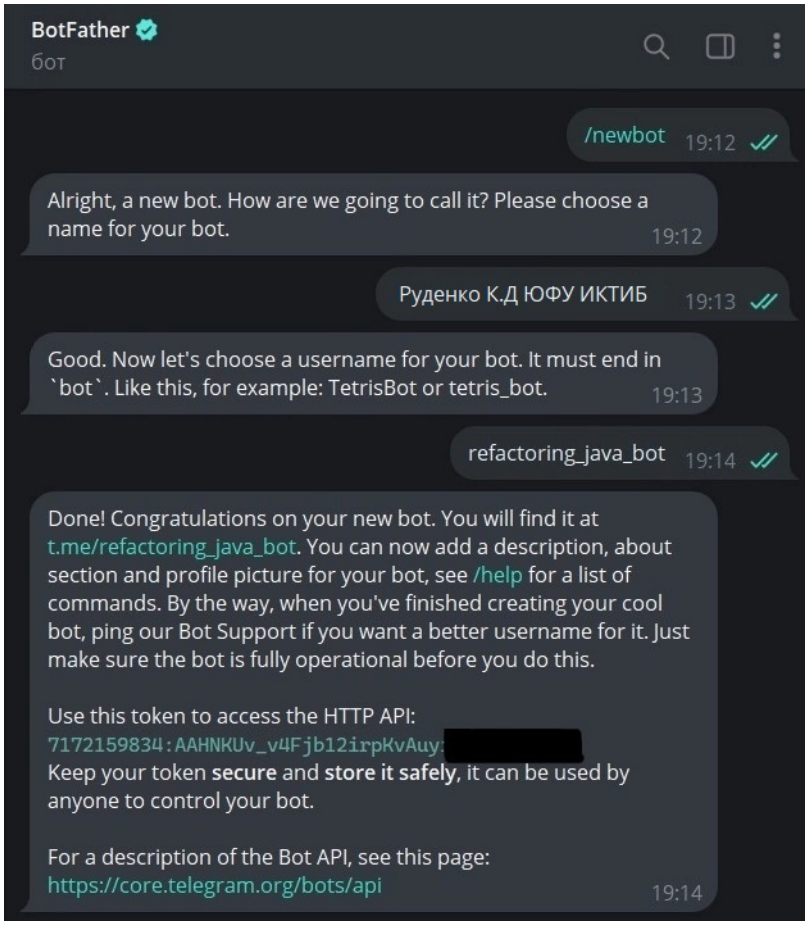


Рис. 12. Создание бота

Команды для полной настройки бота

1) Команда Edit Bot и затем команда Edit About.

2) Отправить текст «Бот создан для помощи оптимизации и рефакторинга java приложений».

3) Команда Edit Description и отправить текст «Привет! Присылай мне код java целиком и я скажу свое мнение по нему!».

4) Команда Edit Description Picture и приветственное фото (рисунок 13).

5) Для установки аватарки бота — команда Edit Botpic и отправить картинку, как показано на рисунке 13 вправом вверхнем углу.

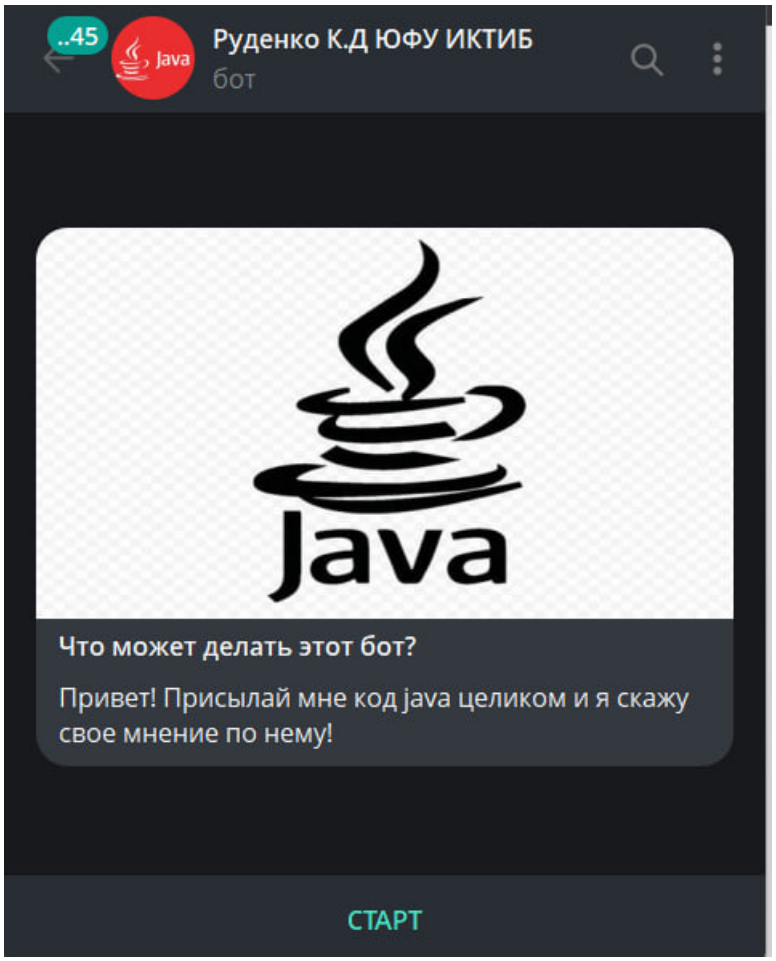


Рис. 13. Экран приветствия

После создания и регистрации бота возможно начать работу с ним. Команда «/start» запускает приветственное сообщение и руководство по работе с ботом (рисунок 13).

4.1. Методы рефакторинга в теллеграм боте

При нажатии на кнопу Методы рефакторинга бот задаст вопрос — О каком методе имя клиента хочет узнать?
и на выбор предлагает рассказать о трех методах, показанных на рисунке 14.

В любой момент существует возможность вернуться в основное меню.

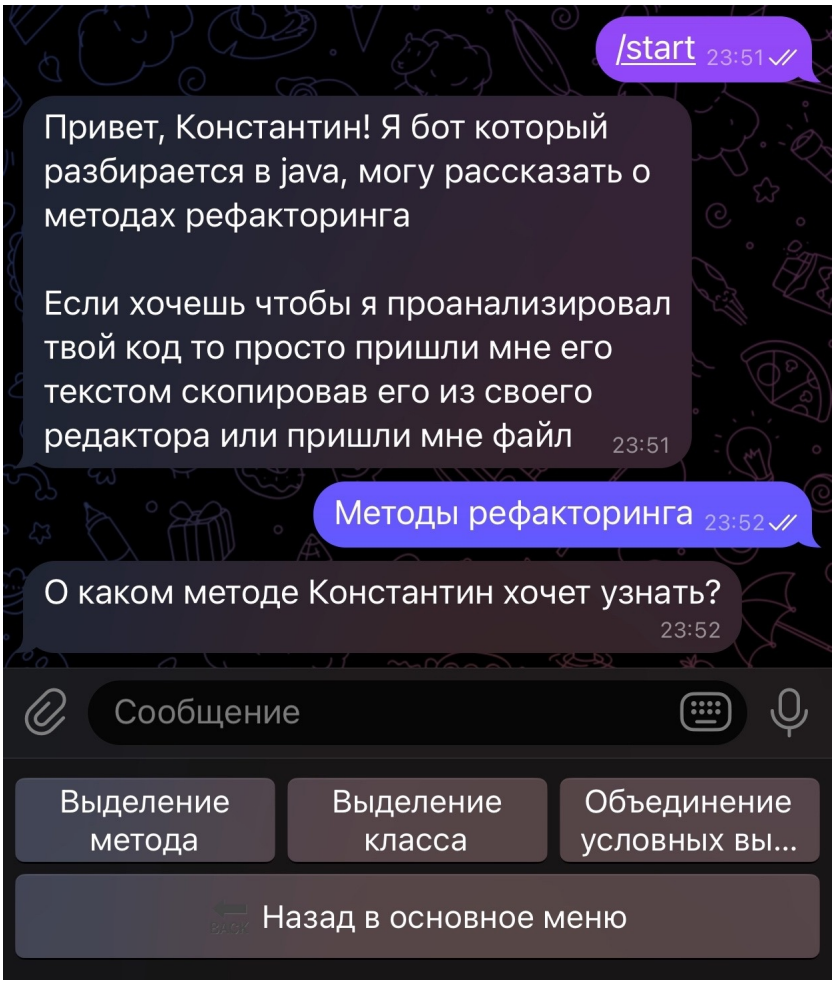


Рис. 14. Методы рефакторинга в телеграмм боте

При нажатии на кнопку выбранного метода, в ответ будет получено определение метода и примеры с неправильной реализацией. И после применения данного метода рефакторинга, также возможно еще изучить достоинства, порядок и причины рефакторинга (рисунок 15 и 16).

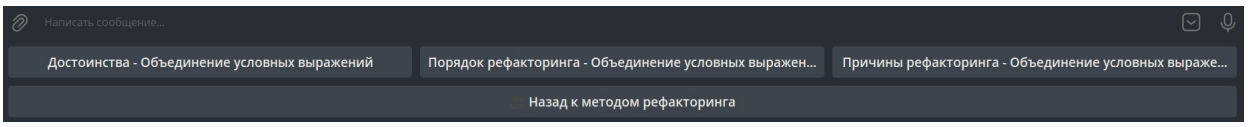


Рис. 15. Кнопки — достоинства, порядок и причины рефакторинга

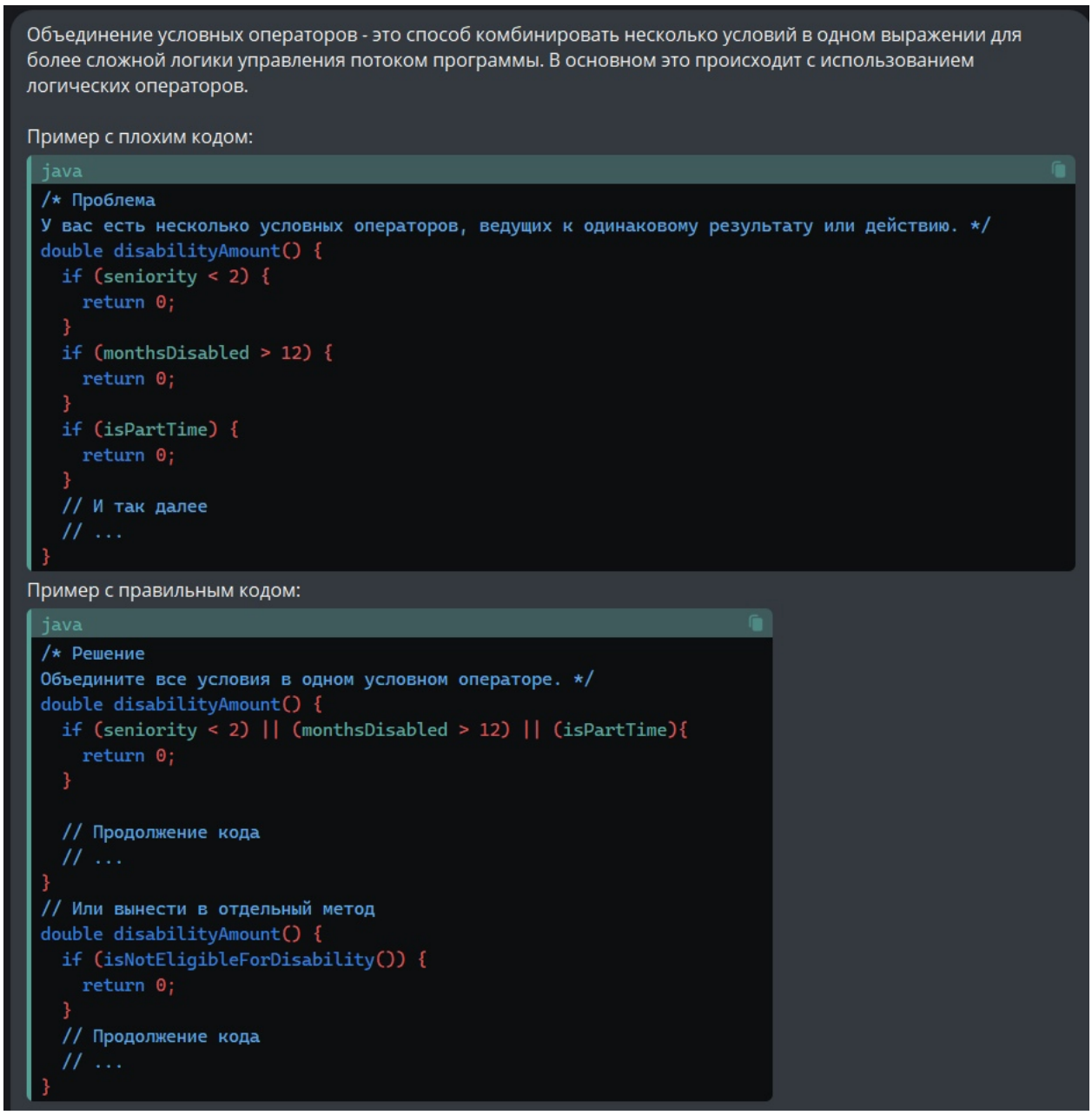


Рис. 16. Нажатие кнопки «Объединение условных выражений»

Бот предоставляет всю подробную информацию по каждому методу, так что после просмотра и изучения можно применять полученные знания для разработки собственных программ на Java.

4.2. Анализ кода в телеграмм боте

При получении текстового сообщения или документа от пользователя через Telegram, бот понимает что ему необходимо это проанализировать. Он проверяет тип контента полученного сообщения. Если это документ, то бот проверяет расширение файла. Если файл имеет расширение «.java», то бот продолжает анализ, в противном случае отправляет пользователю сообщение о том, что файл не может быть проанализирован, так как он принимает только файлы на языке Java как показано на рисунке 16 —

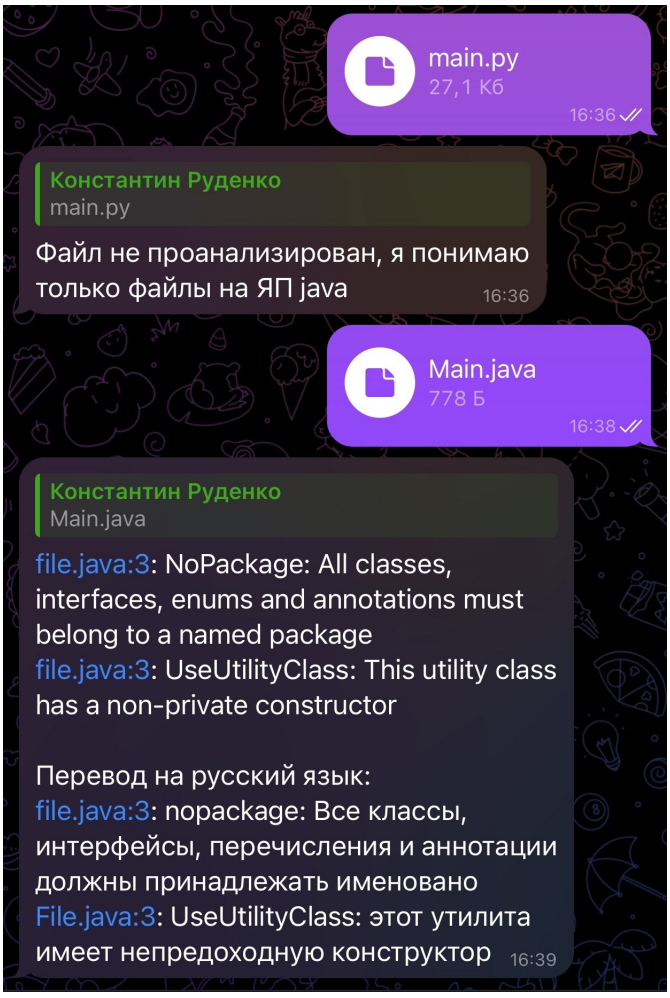


Рис. 17. Отправка разных файлов

в первом случае был прислан файл с расширением «.py» и в ответ бот отослал сообщение «Файл не проанализирован, я понимаю только файлы на ЯП java», а во второй раз пользователь уже отправил файл с кодом

который бот может проанализировать. Данный бот ориентирован на анализ программ на ЯП Java.

Если файл прошел проверку, бот загружает его и сохраняет локально на сервере. Бот вызывает внешний инструмент PMD (Project Mess Detector) для анализа загруженного файла Java. PMD сканирует код на наличие потенциальных проблем и стилевых ошибок, используя заданный набор правил. Результаты анализа кода, полученные от PMD, затем переводятся с помощью Google Translate на русский язык, чтобы сделать их более понятными для пользователя. Наконец, бот отправляет сообщение пользователю с результатами анализа, включая оригинальный текст на английском языке и его перевод на русский язык. Полный код чат-бота представлен в приложении (листинг 5).

4.3. Тестирование анализатора в теллеграм боте

Для проведения тестового рефакторинга я возьму свой проект, который выполнял на производственной практике в «СберТех». Суть задания заключалось в следующем — автоматическая рассылка об уведомлении клиентов банка о завершении срока действий их банковских карт. В результате работы получилось 4 файла: bank_card.java, Client.java, Mail.java, Main.java. Для проверки на необходимость рефакторинга эти файлы отправляюся в диалог бота (рисунок 17). Бот будет анализировать каждый файл добавляя их в очередь, сохраняя и обрабатывая каждый файл по отдельности.

Из ответа видно, что ошибки не критичны, но они допускают нарушения соглашений Java. Файл «bank_card.java» содержит несколько замечаний, которые следует учесть и их можно отнести к следующем категориям:

• Нарушение соглашений о наименовании класса: имя класса «bank_card» не соответствует соглашениям об именовании. Рекомендуется использовать имя, начинающееся с заглавной буквы.

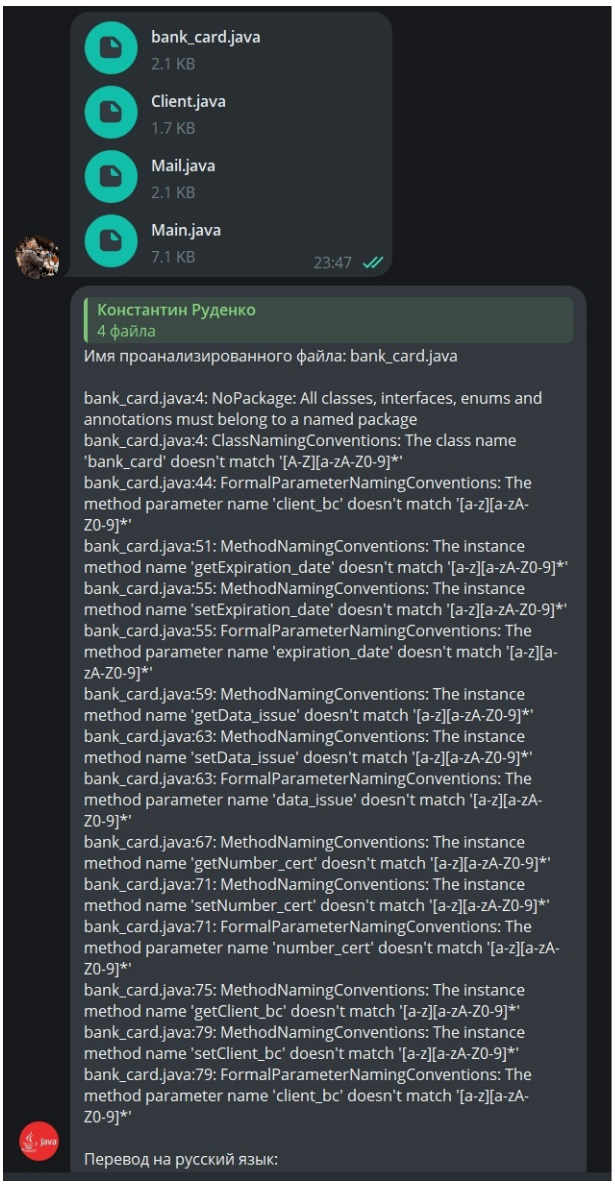


Рис. 18. Ответ на файл «bank_card.java»

• Отсутствие именованного пакета: все классы должны принадлежать именованному пакету. Необходимо добавить директиву package в начало файла.

• Нарушение соглашений о наименовании методов и параметров: названия методов и параметров не соответствуют соглашениям об именовании. Рекомендуется использовать имена, начинающиеся с маленькой буквы, и следующие за ними слова начинать с заглавной буквы.

• Отсутствие информации о пакете: файл не содержит информации о том, к какому пакету относится. Необходимо указать корректный пакет.

Ответ на следующие запросы, которые стоят в очереди бота поступает на файл «Client.java» представленный на рисунке 18, затем «Mail.java» представленный на рисунке 19, и последний самый большой ответ на файл «Main.java» отображен на рисунке 20.

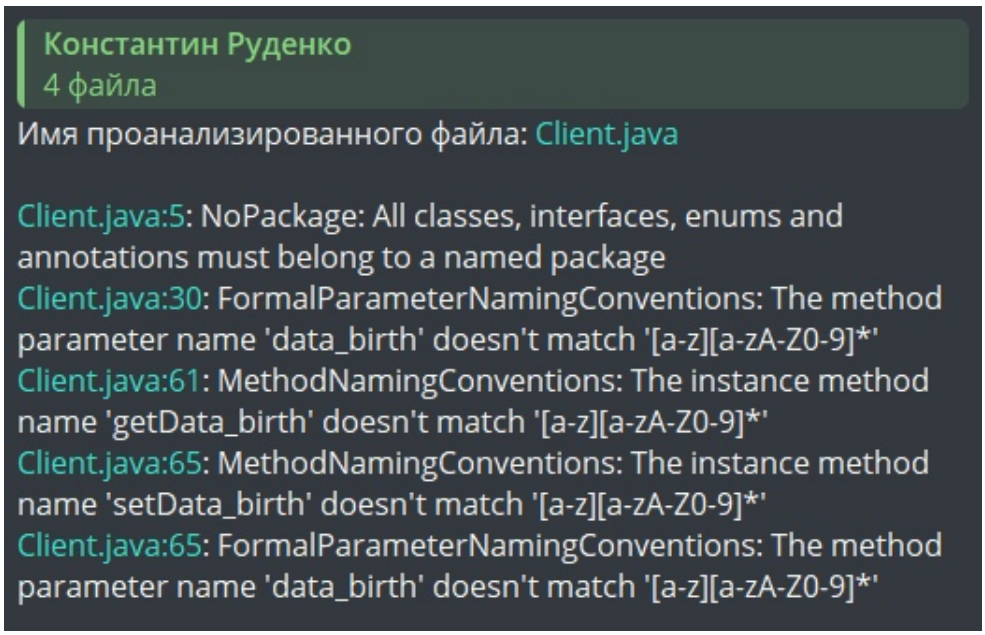


Рис. 19. Ответ на файл «Client.java»

Файл Client.java также содержит несколько замечаний:

• Отсутствие именованного пакета: как и в предыдущем файле, здесь также отсутствует указание на принадлежность к какому-либо пакету.

• Нарушение соглашений о наименовании параметра: имя параметра «data_birth» не соответствует соглашениям об именовании. Рекомендуется использовать имена, начинающиеся с маленькой буквы, и следующие за ними слова начинать с заглавной буквы.

• Нарушение соглашений о наименовании методов и параметров: имена методов и параметров также не соответствуют соглашениям об именовании. Рекомендуется использовать имена, начинающиеся с маленькой буквы, и следующие за ними слова начинать с заглавной буквы.

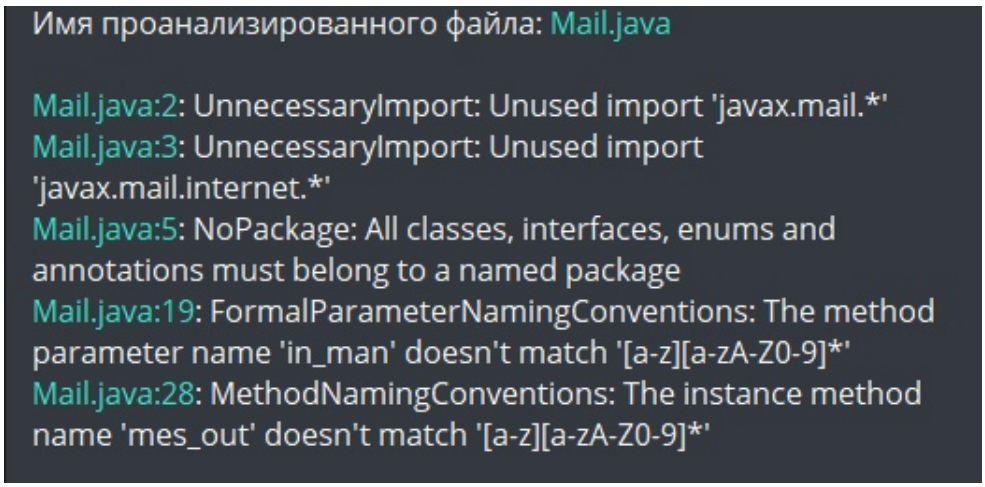


Рис. 20. Ответ на файл «Mail.java»

Файл Mail.java содержит следующие замечания:

• Неиспользуемые импорты: в файле присутствуют неиспользуемые импорты javax.mail.* и javax.mail.internet.*. Рекомендуется удалить их, чтобы избежать лишнего загромождения кода.

• Отсутствие именованного пакета: как и в предыдущих файлах, здесь также отсутствует указание на принадлежность к какому-либо пакету.

• Нарушение соглашений о наименовании параметра: имя параметра in_man не соответствует соглашениям об именовании. Рекомендуется использовать имена, начинающиеся с маленькой буквы, и следующие за ними слова начинать с заглавной буквы.

• Нарушение соглашений о наименовании метода: имя метода mes_out также не соответствует соглашениям об именовании. Рекомендуется использовать имена, начинающиеся с маленькой буквы, и следующие за ними слова начинать с заглавной буквы.

Файл Main.java содержит следующие замечания:

• Отсутствие именованного пакета: как и в предыдущих файлах, здесь также отсутствует указание на принадлежность к какому-либо пакету. Рекомендуется организовать код в соответствии с иерархией пакетов.

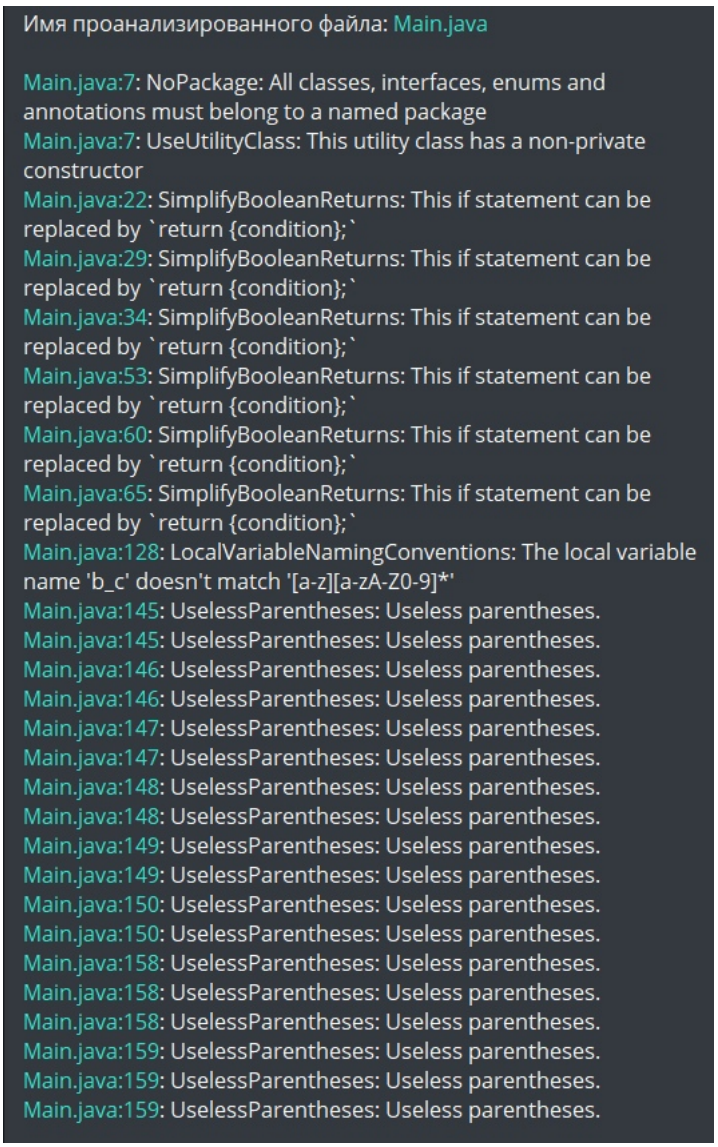


Рис. 21. Ответ на файл «Main.java»

• Использование класса-утилиты с непубличным конструктором: Класс-утилита Main имеет непубличный конструктор. Это может быть проблемой, если класс не предназначен для создания экземпляров. Рекомендуется сделать конструктор приватным или пометить класс как final, если он не должен наследоваться.

• Упрощение возврата булевых значений: несколько условных операторов if можно упростить, заменив их на простые возвращения булевых значений.

• Нарушение соглашений о наименовании локальной переменной: имя локальной переменной b_c не соответствует соглашениям об именовании.

Рекомендуется использовать имена, начинающиеся с маленькой буквы, и следующие за ними слова начинать с заглавной буквы.

• Лишние скобки: в некоторых местах кода присутствуют лишние скобки, которые не влияют на его выполнение и могут быть удалены.

Эти замечания представляют собой стандартные проблемы стиля и организации кода, которые следует исправить для повышения его читаемости и поддерживаемости. Ошибки в этих четырех файлах аналогичны, за исключением некоторых. Прежде чем представлять эти файлы в реализацию, следует внести эти корректировки в соответствии с ответом анализатора.

Заключение

В ходе работы были изучены основные принципы оптимизации и рефакторинга программного кода на языке программирования Java. Основные из них это: повышение эффективности использования ресурсов; улучшение читаемости и поддерживаемости кода; удаление дублирующегося кода; упрощение сложных конструкций; улучшение архитектуры приложения; реконструкция избыточных зависимостей и плохо структурированных модулей; идентификация узких мест производительности; выявление потенциальных ошибок; антипаттерны для улучшения структуры кода.

В рамках работы был разработан телеграмм-бот с функционалом помощи в изучении основных методов рефакторинга. Бот предоставляет примеры и подробные объяснения различных техник рефакторинга кода Java, а также поддерживает анализ кода через текстовые сообщения и отправку большого количества файлов. Телеграмм-бот прошёл комплексное тестирование, включая функциональные и пользовательские тесты, продемонстрировав стабильную работу и правильность выполнения всех заявленных функций, предоставляя пользователям полезную информацию и поддержку в рефакторинге кода.

Таким образом, все задачи, поставленные в рамках ВПК-8. Проект «Прикладное решение на языке Python», 2025, были выполнены.

Литература

1. Герберт Шилдт. Java. Полное руководство. Диалектика-Вильямс. 2018.
2. Кей Хорстманн. Java. Библиотека профессионала. Диалектика-Вильямс. 2020.
3. Bill Phillips, Chris Stewart, Brian Hardy. Android Programming: The Big Nerd Ranch Guide. Big Nerd Ranch Guides. 2019.
4. Dawn Griffiths, David Griffiths. Head First Android Development. O'Reilly Media. 2017.
5. Джошуа Блох. Java. Эффективное программирование. Диалектика-Вильямс. 2019.
6. Брайан Гетц, Тим Пайерлс, Джошуа Блох. Java Concurrency на практике. Питер. 2020.
7. Мартин Фаулер, Кента Бека, Джона Бранта, Уильяма Апдайка и Дона Робертса. рефакторинг Улучшение существующего кода Санкт-Петербург. 2021.
8. Приёмы рефакторинга: [сайт], 2014-2024, URL. https://refactoring.guru/ru/refactoring/techniques (дата обращения: 24.9.2025).
9. Что такое рефакторинг кода: [сайт], Skypro 2024, URL. https://sky.pro/media/chto-takoe-refaktoring-koda/ (дата обращения: 13.10.2025).
10. TIOBE Index for May 2024: [сайт], Skypro 2024, URL. https://www.tiobe.com/tiobe-index/ (дата обращения: 03.10.2025).
11. Как устроен рефакторинг в Java: [сайт], JavaRush 2024, URL. https://javarush.com/groups/posts/refaktoring-java (дата обращения: 17.09.2025).

Приложение

Листинг 2. Выделение класса

```
// Имеется класс Human, в котором также содержится адрес
// проживания и метод, предоставляющий полный адрес
class Human {
   private String name;
   private String age;
   private String country;
   private String city;
   private String street;
   private String house;
   private String quarter;

   public String getFullAddress() {
       StringBuilder result = new StringBuilder();
       return result
                       .append(country)
                       .append(", ")
                       .append(city)
                       .append(", ")
                       .append(street)
                       .append(", ")
                       .append(house)
                       .append(" ")
                       .append(quarter).toString();
   }
}

// Хорошим тоном будет вынести информацию об адресе и метод
// ( поведение обработки данных) в отдельный класс:
class Human {
   private String name;
   private String age;
   private Address address;

   private String getFullAddress() {
       return address.getFullAddress();
   }
}
class Address {
   private String country;
   private String city;
```

```
    private String street;
    private String house;
    private String quarter;

    public String getFullAddress() {
        StringBuilder result = new StringBuilder();
        return result
                        .append(country)
                        .append(", ")
                        .append(city)
                        .append(", ")
                        .append(street)
                        .append(", ")
                        .append(house)
                        .append(" ")
                        .append(quarter).toString();
    }
}
```

Листинг 3. Выделение метода

```
public void calcQuadraticEq(double a, double b, double c) {
    double D = b * b - 4 * a * c;
    if (D > 0) {
        double x1, x2;
        x1 = (-b - Math.sqrt(D)) / (2 * a);
        x2 = (-b + Math.sqrt(D)) / (2 * a);
        System.out.println("x1 = " + x1 + ", x2 = " + x2);
    }
    else if (D == 0) {
        double x;
        x = -b / (2 * a);
        System.out.println("x = " + x);
    }
    else {
        System.out.println("Equation has no roots");
    }
}

// Вынес вычисление всех трех возможных вариантов в отдельные методы

public void calcQuadraticEq(double a, double b, double c) {
    double D = b * b - 4 * a * c;
    if (D > 0) {
        dGreaterThanZero(a, b, D);
```

```
    }
    else if (D == 0) {
        dEqualsZero(a, b);
    }
    else {
        dLessThanZero();
    }
}

public void dGreaterThanZero(double a, double b, double D) {
    double x1, x2;
    x1 = (-b - Math.sqrt(D)) / (2 * a);
    x2 = (-b + Math.sqrt(D)) / (2 * a);
    System.out.println("x1 = " + x1 + ", x2 = " + x2);
}

public void dEqualsZero(double a, double b) {
    double x;
    x = -b / (2 * a);
    System.out.println("x = " + x);
}

public void dLessThanZero() {
    System.out.println("Equation has no roots");
}
```

Листинг 4. Передача всего объекта

```
// Плохой код
public void employeeMethod(Employee employee) {
    // Некоторые действия
    double yearlySalary = employee.getYearlySalary();
    double awards = employee.getAwards();
    double monthlySalary = getMonthlySalary(yearlySalary,
    awards);
    // Продолжение обработки
}

public double getMonthlySalary(double yearlySalary,
    double awards) {
     return (yearlySalary + awards)/12;
}

// Правильная реализация
public void employeeMethod(Employee employee) {
```

```
    // Некоторые действия
    double monthlySalary = getMonthlySalary(employee);
    // Продолжение обработки
}

public double getMonthlySalary(Employee employee) {
    return (employee.getYearlySalary() +
    employee.getAwards())/12;
}
```

Листинг 5. main.py

```
import telebot, subprocess
from telebot.types import Message
from googletrans import Translator
from telebot import types

token = '7172159834:AAHNKUv_v4Fjb12irpKvAuyipwtHn2B4AYw'

bot = telebot.TeleBot(token)


def trans_later(eng_text):
    translator = Translator()
    tr = translator.translate(eng_text, dest="ru")
    return tr.text


def analiz():
    return (subprocess.run(
        "pmd.bat check -d file.java -R rulesets/java/"
        "quickstart.xml -f text",
        shell=True, capture_output=True, text=True,
        encoding='cp866').stdout)


def analiz_itog(message):
    with open('file.java', 'w', encoding='utf-8') as file:
        file.write(message.text)
    analizator_text = analiz() + '\n'
    analizator_text += ("Перевод на русский язык:" + '\n' +
                        trans_later(analizator_text))
    bot.reply_to(message, analizator_text)
```

```
def print_file(f_name):
    with open(f_name, 'r', encoding='utf-8') as file:
        content = file.read()
    return content


@bot.message_handler(commands=['start'])
def start(message):
    markup = types.ReplyKeyboardMarkup(resize_keyboard=True)
    btn1 = types.KeyboardButton("Методы рефакторинга")
    btn2 = types.KeyboardButton(" Задать вопрос")
    markup.add(btn1, btn2)
    bot.send_message(message.chat.id,
     text="Привет, {0.first_name}! Я бот который разбирается "
          "в java, могу рассказать о методах рефакторинга \n\n"
          "Если хочешь чтобы я проанализировал твой код то "
          "просто пришли мне его текстом скопировав его из "
          "своего редактора или пришли мне файл".format(
         message.from_user), reply_markup=markup)


@bot.message_handler(content_types=['document'])
def get_text_messages(message: Message):
    file_info = bot.get_file(message.document.file_id)
    downloaded_file = bot.download_file(file_info.file_path)
    if message.document.file_name[-5:] == '.java':
        src = f"{message.document.file_name}"
        with open(src, 'wb') as new_file:
            new_file.write(downloaded_file)
        analizator_text = analiz(src) + '\n'
        analizator_text += ("Перевод на русский язык:" +
                    '\n' + trans_later(analizator_text))
        if len(analizator_text) > 4096:
            name_file2 = (message.document.file_name[:-5]
                        + "анализ_.txt")
            with open(name_file2, 'w',
                encoding='utf-8') as new_file1:
                new_file1.write(analizator_text)
            with open(name_file2, 'rb') as file3:
                bot.send_document(message.chat.id,
                        file3, caption='Ответ '
'анализа вашего файла больше 4096 символом,'
       ' поэтому отправляю Вам файл с даннными')
        else:
            bot.reply_to(message,
```

```
 "Имя проанализированного файла: "
+ message.document.file_name +
              '\n' + '\n' + analizator_text)
    else:
        bot.reply_to(message,
"Файл не проанализирован,"
" я понимаю только файлы на ЯП java")


@bot.message_handler(content_types=['text'])
@bot.edited_message_handler(content_types=['text'])
def get_text_messeges(message: Message):
    if ((message.text == "Методы рефакторинга") or
            (message.text == " Назад к методом рефакторинга")):
        markup = (types.ReplyKeyboardMarkup
                  (resize_keyboard=True))
        btn1 = types.KeyboardButton("Выделение метода")
        btn2 = types.KeyboardButton("Выделение класса")
        btn3 = types.KeyboardButton(""
                        "Объединение условных выражений")
        back = types.KeyboardButton(" Назад в основное меню")
        markup.add(btn1, btn2, btn3, back)
        bot.send_message(message.chat.id,
text="О каком методе {0.first_name} хочет узнать?".
     format(message.from_user),reply_markup=markup)
    elif (message.text == "Выделение метода"):
        markup = (types.ReplyKeyboardMarkup
                  (resize_keyboard=True))
        btn5 = (types.KeyboardButton
                ("Причины рефакторинга - выделение метода"))
        btn1 = (types.KeyboardButton
                ("Достоинства - выделение метода"))
        btn2 = (types.KeyboardButton
                ("Порядок рефакторинга - выделение метода"))
        back = (types.KeyboardButton
                (" Назад к методом рефакторинга"))
        markup.add(btn1, btn2, btn5, back)
        str_out_false = (print_file
                         ("method_selection_false.java"))
        str_out = print_file("method_selection.java")
        bot.send_message(message.chat.id,
f"Выделение метода \\- это разделиние сложных методов на"
f" более мелкие, каждый из которых выполняет отдельную задачу\\.\n\n"
 f"Пример с плохим кодом\\:"
 f"```java\n{str_out_false}\n```"
```

```
 f"Пример с правильным кодом\\:"
 f"```java\n{str_out}\n```", parse_mode='MarkdownV2',
 reply_markup=markup)
    elif (message.text == "Порядок рефакторинга "
                          "- выделение метода"):
        markup = (types.ReplyKeyboardMarkup
                  (resize_keyboard=True))
        btn1 = types.KeyboardButton("Достоинства "
                                    "- выделение метода")
        btn2 = types.KeyboardButton("Причины рефакторинга
        - выделение метода")
        back = types.KeyboardButton(" Назад "
                            "к методом рефакторинга")
        markup.add(btn1, btn2, back)
        bot.send_message(message.chat.id,
        f"Порядок рефакторинга\\: \n"
f"1\\) Создайте новый метод и назовите его так, "
  f"чтобы название отражало суть того, что будет делать"
                                    f" этот метод\\. \n \n"
f"2\\) Скопируйте беспокоящий вас фрагмент "
                         f"кода в новый метод\\. "
f"Удалите этот фрагмент из старого места и"
           f" замените вызовом вашего нового метода\\.\n"
f"Найдите все переменные, которые"
           f" использовались в этом фрагменте кода\\. "
f"Если они были объявлены внутри этого "
                              f"фрагмента и не используются вне "
f"его, просто оставьте их без изменений \\- они"
f" станут локальными переменными нового метода\\. \n \n"
f"3\\) Если переменные объявлены перед "
                f"интересующим вас участком кода,"
f" значит, их следует передать в"
           f" параметры вашего нового метода, чтобы"
f" использовать значения, которые в них "
                     f"находились ранее\\. "
f"Иногда от таких переменных проще"
                f" избавиться с помощью "
f"замены переменных вызовом метода\\. \n \n"
f"4\\) Если вы видите, что локальная "
                     f"переменная както\\- изменяется "
f"в вашем участке кода, это может означать,"
                f" что её изменённое значение"
f" понадобится дальше в основном методе\\."
           f" Проверьте это\\. "
f"Если подозрение подтвердилось, "
```

```
                f"значение этой переменной следует "
f"возвратить в основной метод, "
                f"чтобы ничего не сломать\\.",
     parse_mode='MarkdownV2', reply_markup=markup)

    elif (message.text == "Причины рефакторинга"
                          " - выделение метода"):
        markup = (types.ReplyKeyboardMarkup
                  (resize_keyboard=True))
        btn1 = types.KeyboardButton("Достоинства - "
                                    "выделение метода")
        btn2 = types.KeyboardButton("Порядок рефакторинга"
                                    " - выделение метода")
        back = types.KeyboardButton(" Назад "
                                    "к методом рефакторинга")
        markup.add(btn1, btn2, back)
        bot.send_message(message.chat.id,
                         f"Причины рефакторинга\\: \n"
f"Чем больше строк кода в методе, тем сложнее разобраться "
                        f"в том, что он делает\\."
f" Это основная проблема, которую решает этот рефакторинг\\. \n \n"
f"Извлечение метода не только убивает множество запашков в коде,"
f" но и является одним из этапов множества других "
f" рефакторингов\\. \n",
parse_mode='MarkdownV2', reply_markup=markup)




    elif (message.text == "Достоинства - выделение метода"):
        markup = types.ReplyKeyboardMarkup(
            resize_keyboard=True)
        btn1 = types.KeyboardButton("Порядок рефакторинга"
                                    " - выделение метода")
        btn2 = types.KeyboardButton("Причины рефакторинга -"
                                    " выделение метода")
        back = types.KeyboardButton(" Назад "
                                    "к методом рефакторинга")
        markup.add(btn1, btn2, back)
        bot.send_message(message.chat.id, f"Достоинства\\: \n"
f"1\\) Улучшает читабельность кода\\. Постарайтесь дать новому"
f" методу название, которое бы отражало суть того, что "
f"он делает\\. Например, createOrder\\(\\), "
f"renderCustomerInfo\\(\\) и т\\. д\\. \n"
f"2\\) Убирает дублирование кода\\. Иногда код,"
```

```
f" вынесенный в метод, можно найти "
f"и в других местах программы\\. В таком случае, "
f"имеет смысл заменить найденные"
f" участки кода вызовом вашего нового метода\\. \n"
f"3\\) Изолирует независимые части кода, уменьшая "
f"вероятность ошибок например\\(,"
f" по вине переопределения не той переменной\\)\\. \n \n ",
     parse_mode='MarkdownV2', reply_markup=markup)
    elif (message.text == " Назад в основное меню"):
        markup = types.ReplyKeyboardMarkup(resize_keyboard=
        True)
        btn1 = types.KeyboardButton("Методы рефакторинга")
        btn2 = types.KeyboardButton(" Задать вопрос")
        markup.add(btn1, btn2)
        bot.send_message(message.chat.id,
text="{0.first_name} сново в основном меню! Напоминаю я"
" бот который разбирается в java, могу рассказать о "
"методах рефакторинга".format(message.from_user),
reply_markup=markup)

    elif (message.text == "Выделение класса"):
        markup = types.ReplyKeyboardMarkup(
            resize_keyboard=True)
        btn5 = types.KeyboardButton("Причины рефакторинга "
                        "- выделение класса")
        btn1 = types.KeyboardButton("Достоинства и"
                    " недостатки - выделение класса")
        btn2 = types.KeyboardButton("Порядок"
                    " рефакторинга - выделение класса")
        back = types.KeyboardButton(" Назад "
                    "к методом рефакторинга")
        markup.add(btn1, btn2, btn5, back)
        photo_false = open('Class_Extraction_False.jpg',
        'rb')
        photo = open('Class_Extraction.jpg', 'rb')
        bot.send_message(message.chat.id,
f"Выделение класса \\(Class Extraction\\) "
f"\\- это процесс рефакторинга кода,"
f" в ходе которого часть функциональности или"
f" данных из одного класса выносится "
f"в отдельный класс\\. "
f"Это делается для того, чтобы каждый класс"
f" имел четко определенную "
f"ответственность и был легче понять,"
f" использовать и поддерживать\\. \n \n"
```

```
f"Пример с неправильной реализацией\\:",
parse_mode='MarkdownV2', reply_markup=markup)
        bot.send_photo(message.chat.id, photo_false)
        bot.send_message(message.chat.id,
f"Пример с правильной реализацией\\:",
parse_mode='MarkdownV2', reply_markup=markup)
        bot.send_photo(message.chat.id, photo)

        elif (message.text == "Порядок рефакторинга -"
                              " выделение класса"):
        markup = types.ReplyKeyboardMarkup(
            resize_keyboard=True)
        btn1 = types.KeyboardButton("Достоинства"
                            " - выделение метода")
        btn2 = types.KeyboardButton("Причины "
            "рефакторинга - выделение метода")
        back = types.KeyboardButton(" Назад "
                        "к методом рефакторинга")
        markup.add(btn1, btn2, back)
        bot.send_message(message.chat.id,
         f"Порядок рефакторинга\\: \n"
f"Перед началом рефакторинга обязательно определите, "
f"как именно следует разделить обязанности класса\\."
f"1\\) Создайте новый класс, который будет содержать "
f"выделенную функциональность\\. \n \n"
f"2\\) Создайте связь между старым и новым классом\\."
f" Лучше всего, если эта связь будет односторонней\\;"
f" при этом второй класс можно будет без проблем "
        f"использовать повторно\\."
f" С другой стороны\\, если вы считаете\\, что это необходимо\\,"
f" всегда можно создать двустороннюю связь\\. \n"
f"3\\) Используйте перемещение поля и перемещение метода для каждого"
f" поля и метода, которые вы решили переместить в новый класс\\."
f" Для методов имеет смысл начинать с приватных\\, таким образом вы"
f" снижаете вероятность допустить массу ошибок\\."
f" Старайтесь двигаться понемногу и тестировать результат после "
f"каждого перемещения\\, это избавит вас от необходимости исправлять"
f" большое число ошибок в самом конце\\. \n \n"
f"После того как с перемещением покончено\\, пересмотрите ещё раз на"
f"   получившиеся классы\\. Возможно\\, старый класс теперь имеет"
f" смысл назвать подругому\\- ввиду его изменившихся обязанностей\\."
f" Проверьте ещё раз\\, можно ли избавиться от двусторонней связи "
f"между классами\\, если она возникла\\."
f"4\\) Ещё одним нюансом является доступность нового класса извне\\."
f" Вы можете полностью спрятать его от клиента\\, сделав приватным\\,"
```

```
f" управляя при этом его полями из старого класса\\. Либо сделать его"
f" публичным\\, предоставив клиенту возможность напрямую менять "
f"значения\\. Решение зависит от того\\, насколько безопасны для "
f"поведения старого класса будут неожиданные прямые изменения"
f" значений в новом классе\\.",
parse_mode='MarkdownV2', reply_markup=markup)

        elif (message.text == "Причины рефакторинга"
                      " - выделение класса"):
        markup = types.ReplyKeyboardMarkup(
        resize_keyboard=True)
        btn1 = types.KeyboardButton("Достоинства "
                        "- выделение метода")
        btn2 = types.KeyboardButton("Порядок "
            "рефакторинга - выделение метода")
        back = types.KeyboardButton(" Назад "
                    "к методом рефакторинга")
        markup.add(btn1, btn2, back)
        bot.send_message(message.chat.id,
     f"Причины рефакторинга\\: \n"
f"Классы всегда изначально выглядят чёткими и понятными\\. "
f"Они выполняют свою работу и не лезут в обязанности "
     f"других классов\\. "
f"Однако, с течением жизни программы добавляется один метод "
f" \\- тут, одно поле  \\- там\\. В результате некоторые классы"
f" получают массу дополнительных обязанностей\\. \n \n",
parse_mode='MarkdownV2', reply_markup=markup)


        elif (message.text == "Достоинства и "
      "недостатки - выделение класса"):
        markup = types.ReplyKeyboardMarkup(
        resize_keyboard=True)
        btn1 = types.KeyboardButton("Порядок"
            " рефакторинга - выделение метода")
        btn2 = types.KeyboardButton("Причины "
            "рефакторинга - выделение метода")
        back = types.KeyboardButton(" Назад "
                "к методом рефакторинга")
        markup.add(btn1, btn2, back)
        bot.send_message(message.chat.id,
         f"Достоинства\\: \n"
f"1\\) Этот рефакторинг призван помочь в соблюдении "
          f"принципа единственной"
f" обязанности класса\\. Это делает код ваших классов"
```

```
          f" очевиднее и понятнее\\.\n"
f"2\\) Классы с единственной обязанностью более"
          f" надёжны и устойчивы к "
f"изменениям\\. Например, у вас есть класс, "
          f"отвечающий за "
f"десять разных вещей\\. И когда вам придётся "
          f"вносить в него изменения, "
f"вы рискуете при корректировках одной вещи "
          f"сломать другие\\. \n\n"
f"Недостатки\\: \n"
f"1\\) Если при проведении этого "
          f"рефакторинга вы перестараетесь,"
f" придётся прибегнуть к встраиванию класса\\. \n",
parse_mode='MarkdownV2', reply_markup=markup)


    # Объединение условных выражений
    elif (message.text == "Объединение условных выражений"):
        markup = types.ReplyKeyboardMarkup(
        resize_keyboard=True)
        btn5 = (types.KeyboardButton
("Причины рефакторинга - Объединение условных выражений"))
        btn1 = (types.KeyboardButton
("Достоинства - Объединение условных выражений"))
        btn2 = (types.KeyboardButton
("Порядок рефакторинга - Объединение условных выражений"))
        back = (types.KeyboardButton
        (" Назад к методом рефакторинга"))
        markup.add(btn1, btn2, btn5, back)
        str_out_false1 = (
    print_file("Consolidate_Conditional_Expression.java"))
        str_out1 = (
    print_file(""
           "Consolidate_Conditional_Expression_false.java"))
        bot.send_message(message.chat.id,
f"Объединение условных операторов \\- это способ комбинировать"
f" несколько условий в одном выражении для более сложной логики"
f" управления потоком программы\\."
f" В основном это происходит с использованием логических "
f"операторов\\. \n \n"
             f"Пример с плохим кодом\\:"
             f"```java\n{str_out1}\n```"
             f"Пример с правильным кодом\\:"
             f"```java\n{str_out_false1}\n```",
 parse_mode='MarkdownV2', reply_markup=markup)
```

```
    elif (message.text == "Причины рефакторинга "
                "- Объединение условных выражений"):
        markup = types.ReplyKeyboardMarkup(
        resize_keyboard=True)
        btn1 = (types.KeyboardButton
        ("Порядок рефакторинга - Объединение условных выражений"))
        btn2 = (types.KeyboardButton
            ("Достоинства - Объединение условных выражений"))
        back = types.KeyboardButton(" Назад"
                            " к методом рефакторинга")
        markup.add(btn1, btn2, back)
        bot.send_message(message.chat.id,
                         f"Причины рефакторинга\\: \n"
f"Код содержит множество чередующихся операторов, которые "
f"выполняют одинаковые действия\\. "
f"Причина разделения операторов неочевидна\\. \n \n"
f"Главная цель объединения операторов - извлечь условие оператора "
f" в отдельный метод или только в один условный оператор, "
f"упростив его понимание\\. \n",
            parse_mode='MarkdownV2', reply_markup=markup)

    elif (message.text ==
          "Достоинства - Объединение условных выражений"):
        markup = types.ReplyKeyboardMarkup(
        resize_keyboard=True)
        btn1 = (types.KeyboardButton
        ("Порядок рефакторинга - Объединение условных выражений"))
        btn2 = (types.KeyboardButton
    ("Причины рефакторинга - Объединение условных выражений"))
        back = types.KeyboardButton(" Назад"
                            " к методом рефакторинга")
        markup.add(btn2, btn1, back)
        bot.send_message(message.chat.id, f"Достоинства\\: \n"
f"1\\) Убирает дублирование управляющего кода\\. Объединение"
f" множества условных операторов, ведущих к одной цели,"
f" помогает показать, что на самом деле вы делаете только "
f"одну сложную проверку, ведущую к одному общему действию\\. \n"
f"2\\) Объединив все операторы в одном, вы позволяете "
f"выделить это сложное условие в новый метод с названием, "
f"отражающим суть этого выражения\\. \n",
 parse_mode='MarkdownV2', reply_markup=markup)

    elif (message.text == "Порядок рефакторинга"
                        " - Объединение условных выражений"):
```

```
        markup = types.ReplyKeyboardMarkup(
        resize_keyboard=True)
        btn1 = types.KeyboardButton("Достоинства"
                    " - Объединение условных выражений")
        btn2 = types.KeyboardButton("Причины рефакторинга"
                        " - Объединение условных выражений")
        back = types.KeyboardButton(""
                            " Назад к методом рефакторинга")
        markup.add(btn2, btn1, back)
        bot.send_message(message.chat.id,
f"Порядок рефакторинга\\: \n"
f"Перед тем как осуществлять рефакторинг, убедитесь, что в"
f" условиях операторов нет «побочных эффектов», или, другими"
f" словами, они не модифицируют чтото\\-, а только возвращают "
f"значения\\."
f"Побочные эффекты могут быть и в коде, который выполняется"
f" внутри самого оператора\\. Например, по результатам условия,"
f" чтото\\- добавляется к переменной\\."
f"1\\) Объедините множество условий в одном с помощью операторов"
f" и и или\\. Объединение операторов обычно следует такому "
f"правилу\\: \n \n"
f"\\-\\-\\-\\- Вложенные условия соединяются с "
f"помощью оператора и\\. \n"
f"\\-\\-\\-\\- Условия, следующие друг за другом, соединяются"
f" с помощью оператора или\\. \n \n"
f"4\\) Извлеките метод из условия оператора и назовите его так,"
f" чтобы он отражал суть проверяемого выражения\\.",
     parse_mode='MarkdownV2', reply_markup=markup)

    else:
        analiz_itog(message)


bot.polling()
```