Лаба 9

Примечание:

Нужно подключить jacoco (это инструмент для измерения покрытия кода тестами), pmd, spotbugs (это статические анализаторы кода).

Если он спросит, то:
1) cobertura по сути не отличается от jacoco
2) раньше использовали findbugs, но с какой-то версии грэдла он стал считаться устаревшим, и вместо него стали использовать spotbugs
3) pmd и spotbugs оба анализируют код на наличие ошибок, просто отчеты внешне отличаются. Я ему сказал, что мне удобнее пользоваться одним из них (забыл уже каким), потому что его отчет мне визуально больше понравился)

А теперь к сути:

PMD

file:///D:/Study/Java%20Optimization/Lab%209/RxJava/build/reports/pmd/main.html

https://docs.gradle.org/current/dsl/org.gradle.api.plugins.quality.PmdExtension.html

1) Прописываешь в build.gradle в плагины:

```
plugins {
    <...> // тут всякие уже подключенные id
    id("pmd") // это дописываешь
}
```

2) Отдельно в build.gradle, не внутри чего-то, прописываешь это:

```
pmd {
    ruleSets = []
    ruleSetFiles = files("pmd.xml")
}
```

Здесь ты задаешь набор правил (он описан в pmd.xml), по которым ты будешь анализировать код. Прикол такой, что в проекте rxjava этот файл с правилами есть, но он нигде не подключен

PMD - это инструмент статического анализа кода, который используется для поиска потенциальных проблем и неправильного использования языка программирования в исходном коде. Он может быть использован для кода на различных языках, но наиболее распространено его применение в Java-проектах. Отчёт PMD помогает разработчикам быстро и эффективно выявлять потенциальные проблемы в их коде, что способствует повышению качества программного обеспечения и облегчает процесс его сопровождения.

Как работает PMD:
PMD сканирует исходные файлы вашего проекта, анализируя их структуру и содержимое. PMD применяет набор правил или шаблонов к исходному коду. Эти правила определяют типичные ошибки, антипаттерны и потенциально опасные конструкции. После применения правил PMD выделяет участки кода, которые соответствуют определенным шаблонам, считая их подозрительными или нуждающимися в дополнительном внимании.
PMD генерирует отчёт, который содержит список найденных проблем, вместе с подробными описаниями и рекомендациями по исправлению.

Отчёт PMD:
Перечень всех найденных проблем в коде + Указание на тип ошибки или антипаттерна, например, неиспользуемая переменная, неправильное форматирование, возможные утечки памяти и т. д. + Ссылки на строки кода, где обнаружены проблемы -> Советы или инструкции по исправлению проблемы.

SpotBugs

file:///D:/Study/Java%20Optimization/Lab%209/RxJava/build/reports/spotbugs/main.xml

https://github.com/spotbugs/spotbugs-gradle-plugin#readme

1) Точно так же прописываешь в плагины:

```
plugins {
    <...> //другие строчки
```

```
    id("com.github.spotbugs") version '4.2.0'
}
```

2) Опять где-нибудь вне всякие штук (я прописал прямо под pmd{<...>}) такое:

```
spotbugs {
    toolVersion = "4.2.2"
    ignoreFailures = false // установить в true, чтобы сборка продолжалась при обнаружении проблем SpotBugs
    reportsDir = file("$buildDir/reports/spotbugs")
}
```

SpotBugs (ранее известный как FindBugs) - это инструмент статического анализа кода для выявления потенциальных ошибок и проблем в Java-приложениях. Он использует анализ байт-кода Java для поиска ошибок, которые могут привести к непредсказуемому поведению, утечкам ресурсов, нарушению соглашений или другим проблемам безопасности.

Как работает SpotBugs:
SpotBugs анализирует скомпилированный байт-код Java, а не исходный код. Это позволяет ему выявлять проблемы, которые могут быть недоступны при обычном анализе исходного кода. SpotBugs применяет набор предопределенных правил (или плагинов) к анализу байт-кода. Эти правила определяют различные виды потенциальных проблем, такие как неправильное использование API, потенциальные утечки ресурсов, возможные ошибки и т. д. После завершения анализа SpotBugs генерирует отчёт, который содержит информацию о найденных проблемах в коде.

Отчёт SpotBugs:
В начале идет перечень классов, которые были загружены в ходе анализа кода. А ниже уже представлена подробная информация о багах. А в самом конце статистика по багам.

Jacoco

http://localhost:63342/rxjava/build/jacocoHtml/index.html

https://docs.gradle.org/6.6.1/userguide/jacoco_plugin.html#sec:jacoco_specific_task_configuration

1) Удостоверься, что в плагинах jacoco уже подключен:

```
plugins {
    <...>
    id("jacoco")
}
```

2) Найди jacocoTestReport и поменяй его на этот:

```
jacocoTestReport {
    dependsOn test
    dependsOn testNG

    reports {
        xml.required = false
        csv.required = false
        html.outputLocation =
layout.buildDirectory.dir('jacocoHtml')
    }
}
```

Здесь ты запрашиваешь отчет в конкретную папку в формате html.
dependsOn test: Эта строка говорит Gradle, что задача jacocoTestReport должна выполниться после выполнения задачи test. Это гарантирует, что данные о покрытии кода будут собраны после выполнения всех тестов.
dependsOn testNG: Эта строка говорит Gradle, что задача jacocoTestReport должна выполниться после выполнения задачи testNG. Предполагается, что testNG - это задача, запускающая тесты с использованием TestNG. Таким образом, она обеспечивает сбор данных о покрытии кода для тестов, запущенных с использованием TestNG.
reports { ... }: Этот блок конфигурирует отчёты, которые будет генерировать jacocoTestReport.
xml.required = false: Здесь отключается генерация XML-отчёта о покрытии кода, поскольку required = false.
csv.required = false: Аналогично, отключается генерация CSV-отчёта о покрытии кода.
html.outputLocation = layout.buildDirectory.dir('jacocoHtml'): Эта строка указывает директорию, в которой будет сохранён HTML-отчёт о покрытии кода. layout.buildDirectory.dir('jacocoHtml') представляет собой динамический путь, который указывает на директорию build/jacocoHtml.

JaCoCo (Java Code Coverage) - это инструмент для измерения покрытия кода тестами в Java-приложениях. Он позволяет определить, какие части вашего

кода были выполнены во время запуска тестов, а какие остались неиспользованными. Это полезный инструмент для оценки качества тестового покрытия вашего приложения и идентификации участков кода, которые нуждаются в дополнительном тестировании.

Как работает JaCoCo:
JaCoCo инструментирует (модифицирует) байт-код вашего приложения, добавляя в него специальные инструкции для отслеживания выполнения каждой строки кода. После инструментирования кода вы запускаете тесты вашего приложения. Во время выполнения тестов JaCoCo собирает данные о том, какие участки кода были выполнены, а какие нет. По завершении выполнения тестов JaCoCo генерирует отчёт о покрытии кода, который показывает процент покрытия каждого класса, метода и строки кода тестами.

Для использования JaCoCo в проекте на Java с помощью системы сборки Gradle, обычно выполняются следующие шаги:

- Добавление плагина JaCoCo в Gradle: Это позволяет настроить задачи для сбора данных о покрытии кода.
- Вы добавляете конфигурацию в ваш build.gradle, чтобы включить сбор данных о покрытии кода при запуске тестов.
- Вы настраиваете задачу в Gradle для генерации отчёта о покрытии кода на основе собранных данных.
- После выполнения тестов вы запускаете задачу для генерации отчёта о покрытии кода.

Отчёт JaCoCo:
Доля кода, выполненного тестами, в процентах + Информация о покрытии каждого класса и метода тестами + Обычно отчёт содержит цветовую дифференциацию строк кода в зависимости от степени их покрытия тестами (например, зеленый для полного покрытия, желтый для частичного и красный для отсутствия покрытия).

Использование JaCoCo помогает вам оценить качество тестового покрытия вашего приложения и идентифицировать участки кода, которые нуждаются в дополнительном тестировании.

1. Missed Instructions (Пропущенные инструкции): Это количество инструкций в коде, которые не были выполнены во время запуска тестов. Инструкции могут быть пропущены, если тесты не покрывают определенные участки кода или если код содержит ветвления или условия, которые не были выполнены во время выполнения тестов.

2. Cov. (Покрытие): Это процент выполненных инструкций от общего количества инструкций в коде. Он показывает, насколько хорошо код покрыт тестами.
3. Missed Branches (Пропущенные ветви): Это количество ветвей в коде, которые не были выполнены во время запуска тестов. Ветви могут быть пропущены, если код содержит условные конструкции (например, if-else), и только одна из ветвей была выполнена во время выполнения тестов.
4. Cxty (Сложность): Это метрика сложности кода, которая показывает, сколько различных путей выполнения кода существует в каждом методе или классе. Чем выше сложность, тем сложнее тестировать и поддерживать код.