5. Übung: Software- und Architekturmetriken für Codequalität und Architekturoptimierung

Überblick und Anwendung einfacher Metriken: Analysieren Sie die grundlegenden Metriken Ihres Codes, wie Methodenanzahl und Parameteranzahl pro Methode, um erste Schwachstellen aufzudecken. Nutzen Sie ein Metrik-Tool wie SonarQube oder ein IDE-Plugin (z. B. IntelliJ Metrics), um die wichtigsten Werte zu erfassen. Ziel: Erste Erkenntnisse über komplexe Codebereiche gewinnen. LLM-Einsatz: Lassen Sie sich mit einem LLM-KI-Tool mögliche Optimierungen oder Vereinfachungen vorschlagen, die durch die Metriken aufgefallen sind (z. B. Methoden zu modularisieren).

Grundlegende Metriken des Package Meditrack (MetricsReloaded):

- Cyclomatic Complexity von 16
- 62 Lines of Code, wovon nur 1 ein Kommentar ist (30%-60% Regel)
- Methodenanzahl = 12; Durchschnittliche Parameteranzahl = 0,5

LLM-Einsatz:

- Dem LLM wurde der Code der zwei Klassen des Packages und die oben gesammelten Metriken übergeben. Ergebnisse:

Zuerst hat das LLM die Metriken interpretiert

🔍 Interpretation der Metriken

Metrik	Bewertung	Bedeutung
Cyclomatic Complexity = 16	etwas hoch für ein so kleines Package	Die drei `validate`-Methoden tragen dazu bei – sie enthalten Bedingungen (`if`) und Exceptions. Kein Problem, aber modularisierbar.
12 Methoden / Ø 0,5 Parameter	normal für ein Modell + Testklasse	Getter dominieren die Methodenanzahl.
Kommentare = 1 von 62 Zeilen	zu niedrig	Deutlich unter der empfohlenen 30–60 %-Kommentarrate – Dokumentation kann verbessert werden.

Dann hat es einen Vorschlag zur Verbesserung der Validierungslogik gemacht

1 Validierungslogik modularisieren

Der Konstruktor ruft drei Validierungsmethoden nacheinander auf.
Wenn die Klasse wächst (mehr Attribute, mehr Prüfungen), wird das schnell unübersichtlich.

Vorschlag:

- Auslagern der Validierung in eine eigene Hilfsklasse, z. B. `PatientValidator`
- Vorteil: Bessere Testbarkeit, klare Trennung von Daten und Logik

java — Code kopieren

```java
public class PatientValidator {
    public static void validate(Patient p) {
        validateName(p.getName());
        validateEmail(p.getEmail());
        validateBirthDate(p.getGeburtsdatum());
    }

    private static void validateName(String name) { ... }
    private static void validateEmail(String email) { ... }
    private static void validateBirthDate(LocalDate geburtsdatum) { ... }
}
```

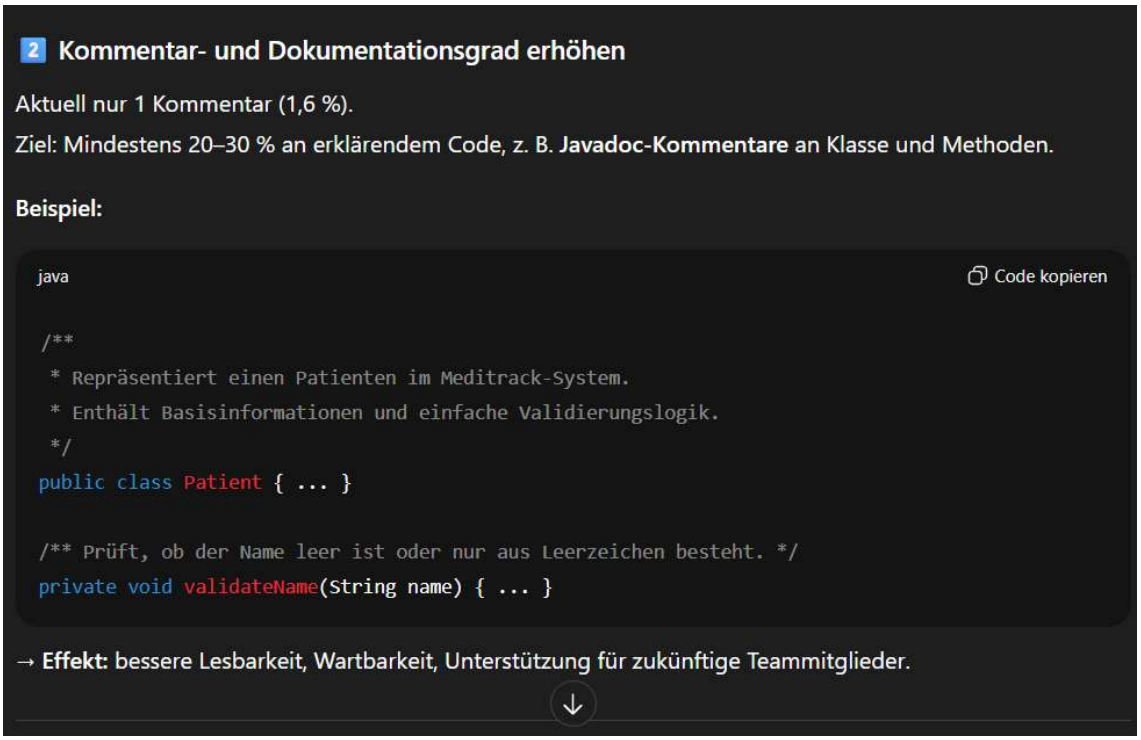
Dann im Konstruktor:

java — Code kopieren

```java
public Patient(String name, String email, LocalDate geburtsdatum) {
    this.id = UUID.randomUUID().toString();
    this.name = name;
    this.email = email;
    this.geburtsdatum = geburtsdatum;
    PatientValidator.validate(this);
}
```

→ **Effekt:** geringere Komplexität pro Klasse, bessere W[illegible]derverwendbarkeit, klare Verantwortlichkeiten.

Als zweites den Hinweis auf die Erhöhung des Anteiles an Kommentaren im Code



2 Kommentar- und Dokumentationsgrad erhöhen

Aktuell nur 1 Kommentar (1,6 %).

Ziel: Mindestens 20–30 % an erklärendem Code, z. B. **Javadoc-Kommentare** an Klasse und Methoden.

Beispiel:

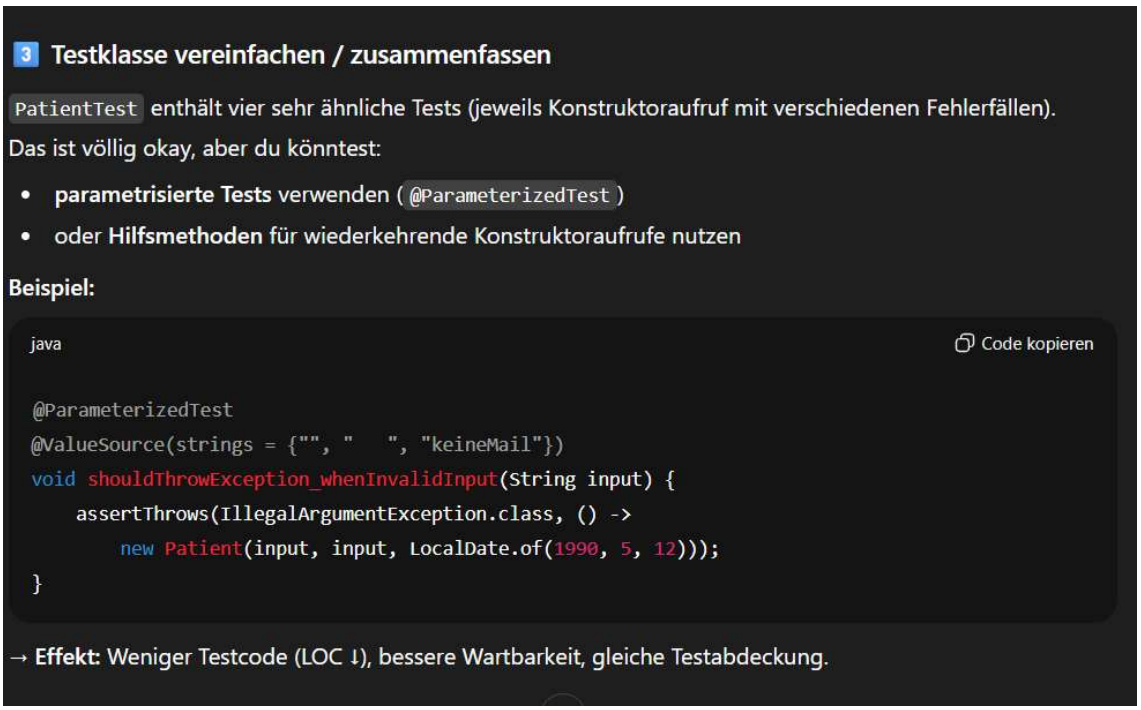
java — Code kopieren

```java
/**
 * Repräsentiert einen Patienten im Meditrack-System.
 * Enthält Basisinformationen und einfache Validierungslogik.
 */
public class Patient { ... }

/** Prüft, ob der Name leer ist oder nur aus Leerzeichen besteht. */
private void validateName(String name) { ... }
```

→ **Effekt:** bessere Lesbarkeit, Wartbarkeit, Unterstützung für zukünftige Teammitglieder.

Vereinfachung der Testklassen



3 Testklasse vereinfachen / zusammenfassen

`PatientTest` enthält vier sehr ähnliche Tests (jeweils Konstruktoraufruf mit verschiedenen Fehlerfällen).

Das ist völlig okay, aber du könntest:

- **parametrisierte Tests** verwenden (`@ParameterizedTest`)
- oder **Hilfsmethoden** für wiederkehrende Konstruktoraufrufe nutzen

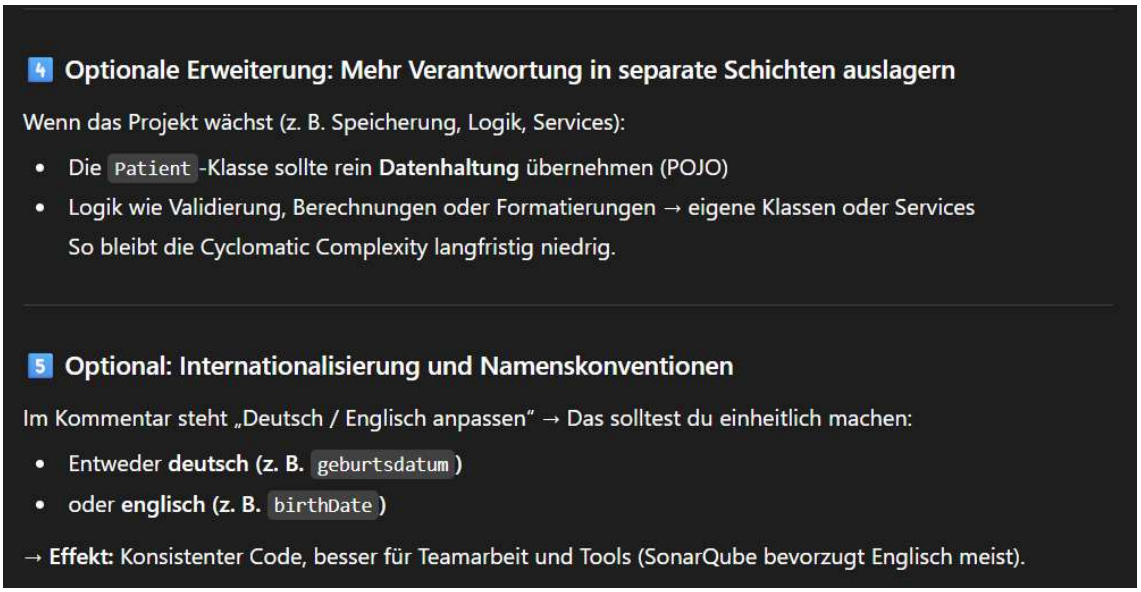
Beispiel:

java — Code kopieren

```java
@ParameterizedTest
@ValueSource(strings = {"", "   ", "keineMail"})
void shouldThrowException_whenInvalidInput(String input) {
    assertThrows(IllegalArgumentException.class, () ->
        new Patient(input, input, LocalDate.of(1990, 5, 12)));
}
```

→ **Effekt:** Weniger Testcode (LOC ↓), bessere Wartbarkeit, gleiche Testabdeckung.

Zwei Optionale Erweiterung



4️⃣ Optionale Erweiterung: Mehr Verantwortung in separate Schichten auslagern

Wenn das Projekt wächst (z. B. Speicherung, Logik, Services):

- Die `Patient`-Klasse sollte rein **Datenhaltung** übernehmen (POJO)
- Logik wie Validierung, Berechnungen oder Formatierungen → eigene Klassen oder Services
  So bleibt die Cyclomatic Complexity langfristig niedrig.

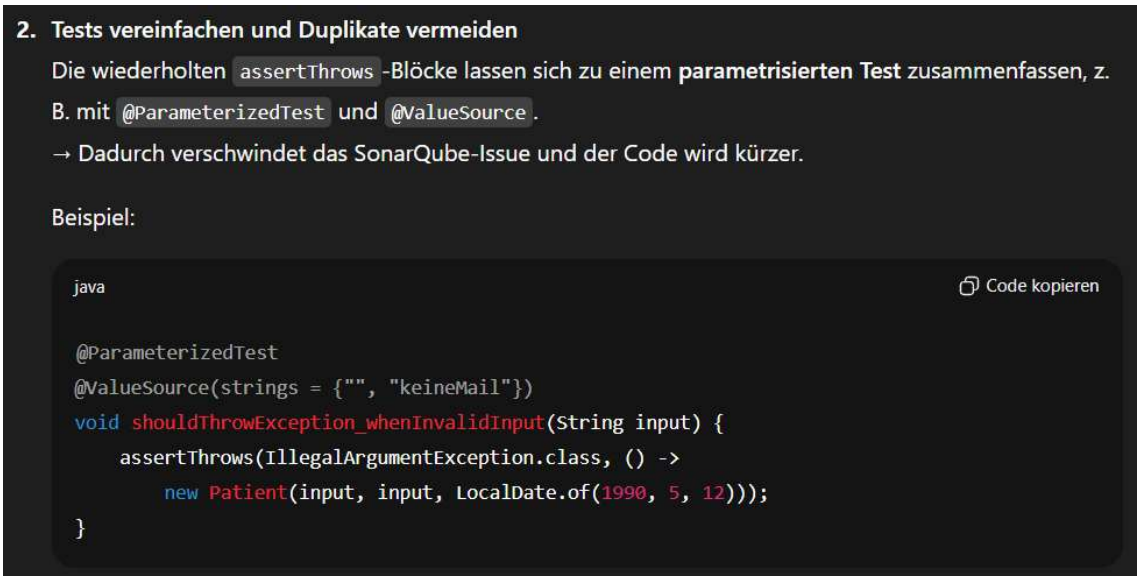
5️⃣ Optional: Internationalisierung und Namenskonventionen

Im Kommentar steht „Deutsch / Englisch anpassen" → Das solltest du einheitlich machen:

- Entweder **deutsch (z. B.** `geburtsdatum`**)**
- oder **englisch (z. B.** `birthDate`**)**

→ **Effekt:** Konsistenter Code, besser für Teamarbeit und Tools (SonarQube bevorzugt Englisch meist).

In Bezug auf die aufgezeigten Probleme von SonarQube schlug das LLM die Verwendung eines parametrisierten Testes:



2. Tests vereinfachen und Duplikate vermeiden

Die wiederholten `assertThrows`-Blöcke lassen sich zu einem **parametrisierten Test** zusammenfassen, z. B. mit `@ParameterizedTest` und `@ValueSource`.

→ Dadurch verschwindet das SonarQube-Issue und der Code wird kürzer.

Beispiel:

java — Code kopieren

```java
@ParameterizedTest
@ValueSource(strings = {"", "keineMail"})
void shouldThrowException_whenInvalidInput(String input) {
    assertThrows(IllegalArgumentException.class, () ->
        new Patient(input, input, LocalDate.of(1990, 5, 12)));
}
```

Test Coverage erweitern und Code Coverage verbessern: Erhöhen Sie die Testabdeckung gezielt für kritische Bereiche Ihrer Anwendung und erweitern Sie bestehende Tests, um eine höhere Testabdeckung (z. B. 80 %) zu erreichen. Verwenden Sie dazu ein Coverage-Tool wie JaCoCo, um Abdeckungswerte zu identifizieren und entwickeln Sie spezifische Tests für komplexe Bereiche. Ziel: Erhöhte Testabdeckung und Sicherstellung der Qualität zentraler Logikkomponenten. LLM-Einsatz: Fragen Sie das LLM nach Testideen oder Edge-Case-Vorschlägen für besonders komplexe Methoden.

JaCoCo konnte folgende Werte identifizieren:
Klasse PatientTest: Class (100%), Method (57%), Line (76%), Branch (100%)
Klasse Patient: Class (100%), Method (75%), Line (90%), Branch (87%)

Zuerst hat das LLM die aktuelle Testabdeckung geschätzt und in einer Tabelle dargestellt.

🔍 1. Aktuelle Testabdeckung (Schätzung anhand deines Codes)

Bereich	Status	Abgedeckt durch
Konstruktor mit gültigen Werten	✅	`shouldCreatePatient_whenValidDataGiven`
Ungültige E-Mail	✅	`shouldThrowException_whenEmailIsInvalid`
Geburtsdatum in der Zukunft	✅	`shouldThrowException_whenBirthDateInFuture`
Leerer Name	✅	`shouldThrowException_whenNameIsEmpty`
Getter (id, name, email, geburtsdatum)	⚠️ Teilweise	nur `getName()` und `getId()` getestet
Null-Werte	❌ Nicht getestet	keiner
Grenzwerte bei Datum (heute, sehr alt)	❌ Nicht getestet	keiner
E-Mail-Formatvarianten	⚠️ Teilweise	nur einfache, aber keine Sonderzeichen
Exception-Messages	❌ Nicht getestet	keiner

Anschließend hat es für Edge-Cases und noch nicht getestete Bereiche jeweils einen Test erstellt und angezeigt:

🧩 (1) Null-Werte testen

Deine `validateName`, `validateEmail` und `validateBirthDate`-Methoden prüfen auf `null`, aber du testest das aktuell nicht.

java — Code kopieren

```java
@Test
void shouldThrowException_whenNameIsNull() {
    assertThrows(IllegalArgumentException.class, () ->
        new Patient(null, "max@example.com", LocalDate.of(1990, 5, 12)));
}

@Test
void shouldThrowException_whenEmailIsNull() {
    assertThrows(IllegalArgumentException.class, () ->
        new Patient("Max", null, LocalDate.of(1990, 5, 12)));
}

@Test
void shouldThrowException_whenBirthDateIsNull() {
    assertThrows(NullPointerException.class, () ->
        new Patient("Max", "max@example.com", null));
}
```

🧩 (2) Gültige Grenzfälle für Geburtsdatum

Um sicherzustellen, dass `validateBirthDate` korrekt arbeitet:

java — Code kopieren

```java
@Test
void shouldAllowBirthDateEqualToToday() {
    assertDoesNotThrow(() ->
        new Patient("Heute", "h@example.com", LocalDate.now()));
}

@Test
void shouldAllowVeryOldBirthDate() {
    assertDoesNotThrow(() ->
        new Patient("Historisch", "h@example.com", LocalDate.of(1900, 1, 1)));
}
```

> 💬 *Erklärung:* Das testet, dass `isAfter(LocalDate.now())` **nur zukünftige** Daten blockiert, nicht heutige oder sehr alte.

🧩 (3) E-Mail-Formate und Randfälle

E-Mail-Validierung ist oft fehleranfällig, daher solltest du einige Varianten testen:

java — Code kopieren

```java
@Test
void shouldAcceptEmailWithSubdomain() {
    assertDoesNotThrow(() ->
        new Patient("Sub", "user@mail.example.com", LocalDate.of(1985, 3, 15)));
}

@Test
void shouldAcceptEmailWithPlusSign() {
    assertDoesNotThrow(() ->
        new Patient("Plus", "user+test@example.com", LocalDate.of(1985, 3, 15)));
}

@Test
void shouldThrowException_whenEmailHasSpaces() {
    assertThrows(IllegalArgumentException.class, () ->
        new Patient("Space", "user name@example.com", LocalDate.of(1985, 3, 15)));
}
```

🧩 (4) Getter-Abdeckung

Aktuell nur `getId()` und `getName()` getestet.

Ergänze Tests, die **alle Getter** abdecken:

java — Code kopieren

```java
@Test
void shouldReturnCorrectValuesFromGetters() {
    LocalDate date = LocalDate.of(1990, 5, 12);
    Patient p = new Patient("Max", "max@example.com", date);

    assertEquals("Max", p.getName());
    assertEquals("max@example.com", p.getEmail());
    assertEquals(date, p.getGeburtsdatum());
    assertNotNull(p.getId());
}
```

(5) Exception Messages prüfen

Das verbessert Lesbarkeit & Wartung (z. B. für Frontend-Validierungen):

java — Code kopieren

```java
@Test
void shouldContainMeaningfulErrorMessage_whenNameEmpty() {
    IllegalArgumentException ex = assertThrows(IllegalArgumentException.class, () ->
        new Patient("", "x@example.com", LocalDate.of(1990, 1, 1)));
    assertTrue(ex.getMessage().contains("Name darf nicht leer sein"));
}
```

Als letztes hat es dann die neue erwartete Coverage in einer Tabelle dargestellt.

3. Erwarteter Nutzen für Coverage

Bereich	vorher	nach neuen Tests
Branch Coverage (if-Zweige)	ca. 60 %	> 90 %
Line Coverage	ca. 70 %	> 85 %
Methodenabdeckung	ca. 75 %	≈ 100 %

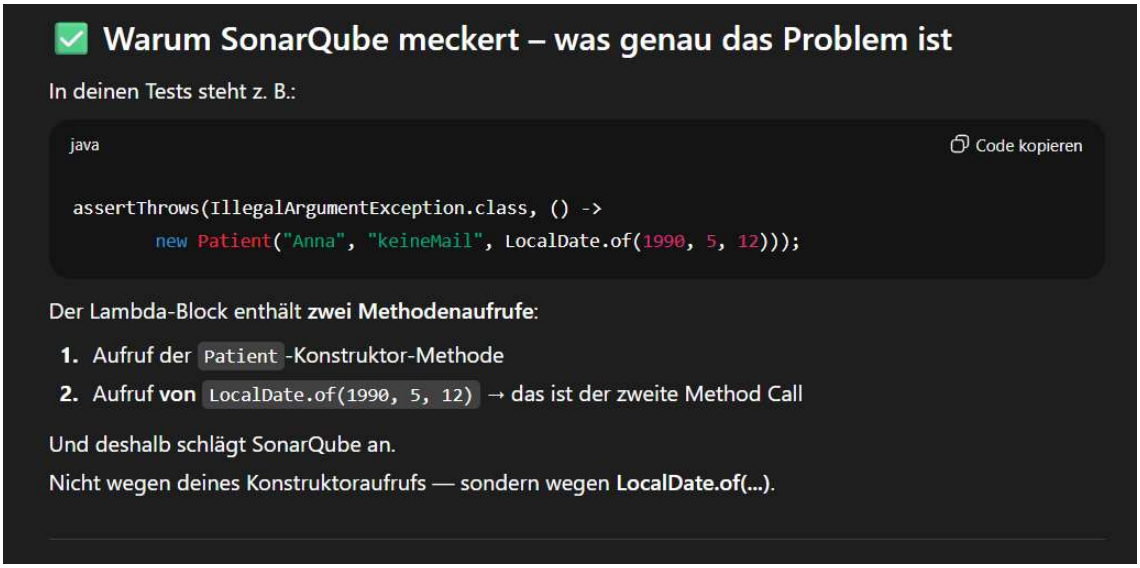
Damit erreichst du problemlos dein Ziel von ≥ **80 % Testabdeckung** laut Aufgabenstellung.
Die Tests decken dann alle logischen Verzweigungen und Randbedingungen ab.

3. Technical Debt und Regelverletzungen mit LLM analysieren: Analysieren Sie mithilfe eines Tools wie SonarQube die technische Schuld und Regelverletzungen in Ihrem Code. Ermitteln Sie dazu technische Schulden in den Bereichen Code-Duplizierung oder veraltete Abhängigkeiten und kategorisieren Sie diese nach Schweregrad. Besprechen Sie mit einem LLM die Ergebnisse und lassen Sie sich Empfehlungen zur Schuldenreduktion geben. Ziel: Reduktion technischer Schulden und Verbesserung der Codequalität. Output: Dokumentation der aktuellen technischen Schuld und Maßnahmen zur Reduzierung, basierend auf LLM-Vorschlägen.

SonarQube:

- in der Klasse PatientTest.java findet SonarQube 3 Maintainability-Issues
- “Refactor the code of the lambda to have only one invocation possibly throwing a runtime exception” in Zeile 18,24 und 30

Das LLM hat daraufhin diese Lösung vorgeschlagen:



✅ Warum SonarQube meckert – was genau das Problem ist

In deinen Tests steht z. B.:

java — Code kopieren

```
assertThrows(IllegalArgumentException.class, () ->
        new Patient("Anna", "keineMail", LocalDate.of(1990, 5, 12)));
```

Der Lambda-Block enthält **zwei Methodenaufrufe**:

1. Aufruf der `Patient`-Konstruktor-Methode
2. Aufruf **von** `LocalDate.of(1990, 5, 12)` → das ist der zweite Method Call

Und deshalb schlägt SonarQube an.

Nicht wegen deines Konstruktoraufrufs — sondern wegen **LocalDate.of(...)**.

1. Email invalid

java — Code kopieren

```
@Test
void shouldThrowException_whenEmailIsInvalid() {
    LocalDate date = LocalDate.of(1990, 5, 12);

    assertThrows(IllegalArgumentException.class,
            () -> new Patient("Anna", "keineMail", date));
}
```

2. Birthdate in the future

java — Code kopieren

```
@Test
void shouldThrowException_whenBirthDateInFuture() {
    LocalDate futureDate = LocalDate.now().plusDays(1);

    assertThrows(IllegalArgumentException.class,
            () -> new Patient("Tom", "tom@example.com", futureDate));
}
```

3. Name empty

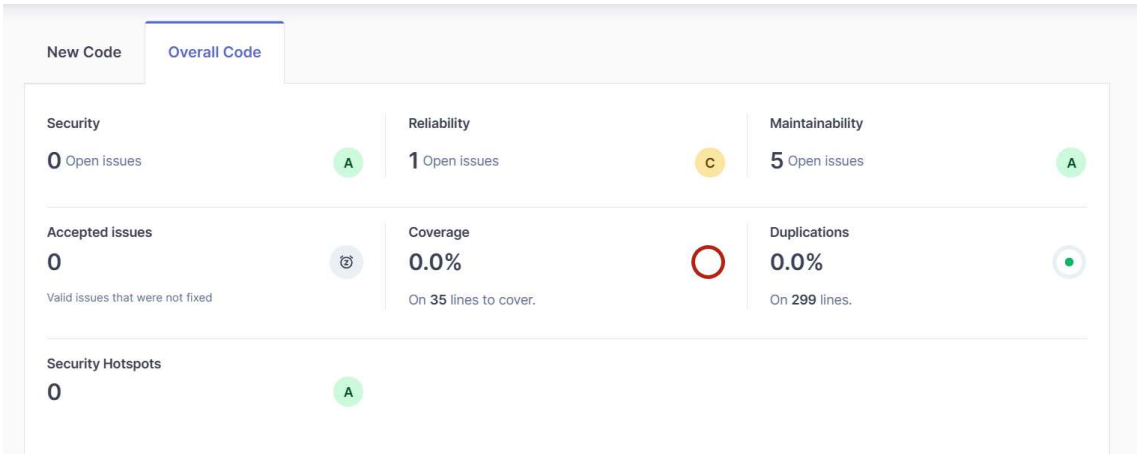
java

Code kopieren

```java
@Test
void shouldThrowException_whenNameIsEmpty() {
    LocalDate date = LocalDate.of(1990, 5, 12);

    assertThrows(IllegalArgumentException.class,
            () -> new Patient("", "tom@example.com", date));
}
```

Die vollständige Analyse des Repositories mit SonarQube ergab die folgende Übersicht:



Hierbei ist anzumerken, dass die Punkte Coverage und Duplications aufgrund fehlender Verknüpfung zu den dafür zuständigen Tools (z.B. Jacoco) bei 0% liegen. Insgesamt wurde nur bei der Klasse UserController.java eine Technical-Debt von 22 Minute berechnet: