

Software Engineering - Code Generierung mit LLMs^[1]

Eine allererste Einführung

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de
Version: 0.1

Folien: [HTML] <https://delors.github.io/se-using-llms/folien.de.rst.html>
[PDF] <https://delors.github.io/se-using-llms/folien.de.rst.html.pdf>
Fehler melden: <https://github.com/Delors/delors.github.io/issues>

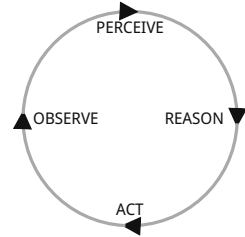
^[1] Erstellt unter Zuhilfenahme von ChatGPT

LLMs und Code-Generierung

Normales LLM

- Antwortet direkt:
Eingabe → Ausgabe
- Schnell, ausreichend
für einfache Aufgaben
- Risiko: unvollständiger
oder fehlerhafter Code bei
komplexeren Problemen
- Beispiel: „Schreibe
eine Python-Funktion
für Fibonacci-Zahlen“
→ sofortiger Code

Thinking-LLM



- Führt interne „Denk“-Schritte vor der Antwort aus
- Reflektiert, prüft und verbessert den Code, bevor er ausgegeben wird
- Langsamer, aber zuverlässiger bei komplexen Aufgaben
- Beispiel: Überlegt Algorithmus-Varianten, Edge-Cases und Tests, bevor Code zurückkommt

Zusammenfassung

Normale LLMs sind schnell für Routinetätigkeiten – Thinking-LLMs liefern bessere Ergebnisse bei schwierigen Programmieraufgaben.

Prompting-Strategien für Code-Generierung mit LLMs

Klare Aufgabenstellung:

Programmiersprache und gewünschte Bibliotheken benennen.

Beispiel

Schreibe eine Funktion in *Python*, die ... mit *NumPy* ...

Struktur vorgeben: Erwünschte Signatur oder Interfaces spezifizieren und dabei möglichst präzise sein.

Beispiel

Implementiere `def sort_list(xs: list[int]) -> list[int]:`

Schrittweise Anleitungen:

Erst Algorithmus erklären lassen, dann den Code generieren.

Hilft bei komplexen Problemen (Divide & Conquer im Prompt).

Beispiele nutzen: „Few-Shot Prompting“ mit kleinen Code-Snippets als Vorlage.

Fördert Einhaltung von Stil, Syntax und Konventionen.

Fehlervermeidung: Auf Testfälle im Prompt hinweisen: „Der Code muss diese Tests bestehen ...“

Iterative Verfeinerung:

Erst groben Entwurf generieren, dann gezielt Verbesserungen anfordern

Beispiel

„Optimiere die Laufzeit“ oder „Füge Typannotationen hinzu“.

Typische Stolperfallen beim Code-Prompting

- Unpräzise Sprache:**
- Keine Programmiersprache/Technologien angegeben → Modell wählt zufällig.
 - Lösung: Sprache und Version im Prompt festlegen

Fehlende Constraints:

- Ohne Funktionssignatur oder Interfaces generiert LLM „freie“ Lösungen
- Lösung: Signaturen, Klassen oder API-Vorgaben angeben

Vermischte Anforderungen:

- Prompt enthält Code + Prosa → Antwort enthält unnötige Erklärungen
- Lösung: explizit nur *Codeblock* anfordern

Zu komplex in einem Schritt:

- Lange Prompts mit vielen Bedingungen führen zu Fehlern
- Lösung: Problem in mehrere Teilaufgaben aufsplitten, dabei jedoch alle für einen Schritt relevanten Anforderungen benennen, um sich (viel) Nacharbeit zu sparen.

Modernere LLMs können immer besser mit komplexen Prompts umgehen und hier sind auch weitere Verbesserungen zu erwarten.

Keine Tests spezifiziert:

- Modell weiß nicht, wann Lösung „korrekt“ ist
- Lösung: Testfälle im Prompt beilegen oder explizit anfordern

Übermäßiges Vertrauen:

- ! Generierter Code wird ohne Überprüfung übernommen!
- ! Lösung: Immer Review, Linting und Tests durchführen. Insbesondere Versionen von externen Bibliotheken überprüfen, diese sind häufig nicht aktuell oder veraltet.

Achtung!

Beim Einbinden von Tools oder Bibliotheken ist die Zielbibliothek immer zu prüfen! Am Besten einen Dependency-Manager verwenden.

Mensch vs. Maschine

[...] Hoher Komplexitätsgrad als Stolperstein

Viele KI-Tools sind mittlerweile so leistungsstark geworden, dass sie Entwicklern einen echten Mehrwert bringen. Komplexe Coding-Aufgaben können sich für Kollegen KI jedoch laut der Studie als Stolperstein herausstellen. Darunter fallen etwa Aspekte wie das Kontextverständnis für sehr umfangreiche Codebasen, höhere logische Komplexitätsgrade sowie die Fähigkeit, Codestrukturen so zu planen und umzusetzen, dass ihre Qualität auch langfristig auf dem gleichen Niveau bleibt. [...]

—2.9.2025 - Heise Online - Das machen Menschen besser: Studie zeigt Grenzen von Coding-Agenten

Es ist davon auszugehen, dass sich diese Situation zeitnahe noch stärker verschieben wird, dadurch, dass die Modelle bei Unsicherheit nachfragen werden.

1. Exemplarische Verwendung von LLMs für die Codegenerierung

Beispielhafte Verwendung von GPT-4.1

Das Ziel ist die Generierung einer Kommandozeilenanwendung zur Berechnung der Quadratwurzel einer Zahl mit vorgegebener, aber beliebiger Genauigkeit.

Achtung!

Der Chat bzw. die Ausgaben wurden teilweise reformatiert und gekürzt, um die Lesbarkeit zu verbessern. Der grundsätzliche Ablauf ist zu 100% erhalten, um die Herausforderungen zu verdeutlichen.

Beispielhafte Verwendung von GPT-4.1

1. Schritt: Grundlegende Methode implementieren lassen.

User

Bitte schreibe mir eine Methode in Java, die die Newton-Raphson-Methode anwendet zur Berechnung der Quadratwurzel.

Agent

Hier ist eine Java-Methode, die mit Newtons Verfahren die Quadratwurzel einer Zahl berechnet:

```
1 class NewtonSqrt {
2     /**
3      * Berechnet die Quadratwurzel einer Zahl mit Newtons Verfahren.
4      * @param n Die Zahl, deren Quadratwurzel berechnet werden soll ( $n \geq 0$ )
5      * @param tolerance Die gewünschte Genauigkeit (z.B.  $1e-10$ )
6      * @return Die berechnete Quadratwurzel von n */
7     public static double sqrt(double n, double tolerance) {
8         if (n < 0) { throw new IllegalArgumentException("n muss  $\geq 0$  sein"); }
9         if (n == 0) return 0;
10        double x = n;
11        while (Math.abs(x * x - n) > tolerance) {
12            x = 0.5 * (x + n / x);
13        }
14        return x;
15    }
16 }
```

Verwendung:

```
1 public class ExampleUsage {
2     public static void main(String[] args) {
3         double wurzel = NewtonSqrt.sqrt(25, 1e-10);
4         System.out.println("Quadratwurzel von 25: " + wurzel);
5     }
6 }
```

Beispielhafte Verwendung von GPT-4.1

2. Schritt: Ermöglichen einer konfigurierbaren Anzahl Stellen.

User

Die Methode soll in der Lage sein eine - ggf. sehr große - Anzahl an Nachkommastellen zu liefern.

Agent

Um eine sehr große Anzahl an Nachkommastellen zu berechnen, solltest du statt `double` die Klasse `BigDecimal` verwenden. Damit kannst du beliebige Genauigkeit erreichen, solange der Speicher reicht. Newtons Verfahren lässt sich auch mit `BigDecimal` umsetzen.

Hier ist eine Methode, die die Quadratwurzel einer Zahl mit Newtons Verfahren und beliebiger Genauigkeit berechnet:

```
1  import java.math.BigDecimal;
2  import java.math.MathContext;
3  import java.math.RoundingMode;
4
5  public class BigDecimalNewtonSqrt {
6      /**
7       * Berechnet die Quadratwurzel einer Zahl mit Newtons Verfahren und hoher Genauigkeit.
8       * @param n Die Zahl, deren Quadratwurzel berechnet werden soll ( $n \geq 0$ )
9       * @param scale Anzahl der gewünschten Nachkommastellen
10      * @return Die berechnete Quadratwurzel von n als BigDecimal
11      */
12     public static BigDecimal sqrt(BigDecimal n, int scale) {
13         if (n.compareTo(BigDecimal.ZERO) < 0) {
14             throw new IllegalArgumentException("n muss  $\geq 0$  sein");
15         }
16         if (n.compareTo(BigDecimal.ZERO) == 0) return BigDecimal.ZERO;
17
18         BigDecimal two = BigDecimal.valueOf(2);
19         MathContext mc = new MathContext(scale + 5, RoundingMode.HALF_UP); // Extra Stellen für
20         BigDecimal x = n;
21         BigDecimal lastX;
22         int maxIterations = 1000;
23         int count = 0;
24
25         do {
26             lastX = x;
27             x = n.divide(x, mc).add(x).divide(two, mc);
28             count++;
29         } while (x.subtract(lastX).abs().compareTo(BigDecimal.ONE.scaleByPowerOfTen(-scale)) > 0);
30
31         return x.setScale(scale, RoundingMode.HALF_UP);
32     }
33 }
```

Beispiel für die Verwendung:

```
1  import java.math.BigDecimal;
2
3  public class BigDecimalNewtonSqrtTest {
4      public static void main(String[] args) {
5          BigDecimal n = new BigDecimal("2");
6          int scale = 50; // 50 Nachkommastellen
7          BigDecimal wurzel = BigDecimalNewtonSqrt.sqrt(n, scale);
```

```
8      System.out.println("Quadratwurzel von 2 mit 50 Nachkommastellen:");
9      System.out.println(wurzel.toPlainString());
10
11 }
```

Mit dieser Methode kannst du die Quadratwurzel mit beliebig vielen Nachkommastellen berechnen. Passe einfach den Wert von `scale` an deine Anforderungen an.

Beispielhafte Verwendung von GPT-4.1

3. Schritt: Umwandlung in ein Kommandozeilenprogramm

User

Bitte stecke alle Methoden in eine Klasse und erweitere die Main Methode so, dass ich die Parameter auf der Kommandozeile eingeben kann. Erzeuge auch eine entsprechende Datei.

Agent

Hier ist die vollständige Java-Klasse `BigDecimalNewtonSqrt` mit Newtons Verfahren für die Quadratwurzel und einem Kommandozeilen-Interface. Die Datei wurde unter `...src/BigDecimalNewtonSqrt.java` erstellt.

```
1  import java.math.BigDecimal;
2  import java.math.MathContext;
3  import java.math.RoundingMode;
4
5  public class BigDecimalNewtonSqrt {
6
7      /**
8       * Berechnet die Quadratwurzel einer Zahl mit Newtons Verfahren und hoher Genauigkeit.
9       * @param n Die Zahl, deren Quadratwurzel berechnet werden soll ( $n \geq 0$ )
10      * @param scale Anzahl der gewünschten Nachkommastellen
11      * @return Die berechnete Quadratwurzel von n als BigDecimal
12      */
13      public static BigDecimal sqrt(BigDecimal n, int scale) {
14          if (n.compareTo(BigDecimal.ZERO) < 0) {
15              throw new IllegalArgumentException("n muss  $\geq 0$  sein");
16          }
17          if (n.compareTo(BigDecimal.ZERO) == 0) return BigDecimal.ZERO;
18
19          BigDecimal two = BigDecimal.valueOf(2);
20          MathContext mc = new MathContext(scale + 5, RoundingMode.HALF_UP); // Extra Stellen für
21          BigDecimal x = n;
22          BigDecimal lastX;
23          int maxIterations = 1000;
24          int count = 0;
25
26          do {
27              lastX = x;
28              x = n.divide(x, mc).add(x).divide(two, mc);
29              count++;
30          } while (x.subtract(lastX).abs().compareTo(BigDecimal.ONE.scaleByPowerOfTen(-scale)) > 0);
31
32          return x.setScale(scale, RoundingMode.HALF_UP);
33      }
34
35      /**
36       * Kommandozeilen-Interface:
37       * Parameter:
38       *   args[0] - Zahl (z.B. "2")
39       *   args[1] - Anzahl der Nachkommastellen (z.B. "50")
40       */
41      public static void main(String[] args) {
42          if (args.length < 2) {
43              System.out.println("Verwendung: java BigDecimalNewtonSqrt <Zahl> <Nachkommastellen>");
44              System.out.println("Beispiel: java BigDecimalNewtonSqrt 2 50");
45              return;
46          }
47      }
48  }
```

```

46     }
47     try {
48         BigDecimal n = new BigDecimal(args[0]);
49         int scale = Integer.parseInt(args[1]);
50         BigDecimal wurzel = sqrt(n, scale);
51         System.out.println("Quadratwurzel von " + n.toString() + " mit " + scale + " N
52         System.out.println(wurzel.toString());
53     } catch (Exception e) {
54         System.out.println("Fehler: " + e.getMessage());
55     }
56 }
57 }

```

Du kannst die Klasse wie folgt kompilieren und ausführen:

```

javac src/BigDecimalNewtonSqrt.java
java -cp src BigDecimalNewtonSqrt 2 50

```

Das berechnet die Quadratwurzel von 2 mit 50 Nachkommastellen. Passe die Parameter nach Bedarf an.

Beispielhafte Verwendung von GPT-4.1

4. Schritt: Kritisches Hinterfragen des Codes

User

In der Methode `sqrt` ist die Anzahl an Iterationen auf Max 1000 gesetzt. Sollte dies nicht abhängig sein von der Anzahl der geforderten Nachkommastellen. Wenn ich zum Beispiel 10000 Nachkommastellen anfordere, dann erwarte ich auch, dass diese präzise sind und dies dürfte bei 1000 Iterationen nicht der Fall sein, oder?

Agent

Das ist ein sehr guter und berechtigter Hinweis! Die maximale Anzahl der Iterationen sollte tatsächlich von der gewünschten Genauigkeit (`scale`) abhängen. Je mehr Nachkommastellen du verlangst, desto mehr Iterationen sind in der Regel nötig, damit Newtons Verfahren konvergiert und die gewünschte Präzision erreicht wird.

Warum?

Newton's Verfahren verdoppelt grob die Anzahl der korrekten Stellen pro Iteration, aber bei sehr hoher Präzision kann es passieren, dass die Differenz zwischen zwei Iterationen so klein wird, dass sie erst nach sehr vielen Durchläufen unter die gewünschte Schwelle fällt.

Empfehlung:

Setze `maxIterations` proportional zur gewünschten Nachkommastellenzahl, z.B.

`maxIterations = Math.max(100, scale * 2)` oder sogar `scale * 5` für sehr hohe Sicherheit.

Hier ist die angepasste Methode:

```
1 public static BigDecimal sqrt(BigDecimal n, int scale) {
2     if (n.compareTo(BigDecimal.ZERO) < 0) {
3         throw new IllegalArgumentException("n muss ≥ 0 sein");
4     }
5     if (n.compareTo(BigDecimal.ZERO) == 0) return BigDecimal.ZERO;
6
7     BigDecimal two = BigDecimal.valueOf(2);
8     MathContext mc = new MathContext(scale + 5, RoundingMode.HALF_UP); // Extra Stellen für Gen
9     BigDecimal x = n;
10    BigDecimal lastX;
11    int maxIterations = Math.max(100, scale * 5); // Dynamisch abhängig von scale
12    int count = 0;
13
14    do {
15        lastX = x;
16        x = n.divide(x, mc).add(x).divide(two, mc);
17        count++;
18    } while (x.subtract(lastX).abs().compareTo(BigDecimal.ONE.scaleByPowerOfTen(-scale)) > 0 &&
19
20    if (count ≥ maxIterations) {
21        throw new ArithmeticException("Maximale Iterationszahl (" + maxIterations + ") erreicht
22    }
23
24    return x.setScale(scale, RoundingMode.HALF_UP);
25 }
```

Was ist neu? - `maxIterations` ist jetzt abhängig von `scale`. - Es wird eine Exception geworfen, falls die Iterationszahl überschritten wird (damit du weißt, dass das Ergebnis evtl. nicht präzise genug ist).

Fazit:

Mit dieser Anpassung kannst du auch sehr viele Nachkommastellen zuverlässig berechnen. Bei extrem hohen Anforderungen kann es trotzdem sinnvoll sein, die Konvergenz zu überwachen und ggf. die Startwerte oder die MathContext-Präzision weiter zu erhöhen.

Beispielhafte Verwendung von GPT-4.1

5. Schritt: Dokumentation verbessern lassen

User

Bitte aktualisiere die Dokumentation der Methode in Hinblick darauf, dass eine `ArithmeticException` geworfen werden kann.

Agent

Die Dokumentation der Methode `sqrt` wurde aktualisiert und enthält jetzt den Hinweis, dass eine `ArithmeticException` geworfen werden kann, falls die maximale Iterationszahl überschritten wird und das Ergebnis möglicherweise ungenau ist.

Hier ist der relevante Ausschnitt:

```
1  /**
2   * Berechnet die Quadratwurzel einer Zahl mit Newtons Verfahren und hoher Genauigkeit.
3   * @param n Die Zahl, deren Quadratwurzel berechnet werden soll ( $n \geq 0$ )
4   * @param scale Anzahl der gewünschten Nachkommastellen
5   * @return Die berechnete Quadratwurzel von n als BigDecimal
6   * @throws ArithmeticException falls die maximale Iterationszahl überschritten wird und das Erg
7   */
8  public static BigDecimal sqrt(BigDecimal n, int scale) {
9      ...
10 }
```

Damit ist die Methode und ihre Dokumentation jetzt korrekt und robust für sehr hohe Präzision.

Beispielhafte Verwendung von GPT-4.1

6. Schritt: Dokumentation weiter verbessern lassen

User

Bitte dokumentiere bis zu wieviel Stellen die Methode aufgrund der Beschränkung des Counters auf den Datentyp `int` maximal präzise sein kann.

Agent

Sehr gerne!

Da der Iterationszähler `count` vom Typ `int` ist, kann die Methode maximal `Integer.MAX_VALUE` Iterationen durchführen. Da die maximale Iterationszahl als `maxIterations = Math.max(100, scale * 5)` gesetzt wird, ist die maximale Anzahl an Nachkommastellen, die du zuverlässig berechnen kannst, durch die Größe des `int`-Datentyps begrenzt.

Berechnung:

- `Integer.MAX_VALUE = 2 147 483 647`
- Die Formel für die Iterationen: `scale * 5 ≤ Integer.MAX_VALUE`
- Daraus folgt: `scale ≤ Integer.MAX_VALUE / 5 ≈ 429 496 729`

Dokumentationsvorschlag für die Methode:

```
1  /**
2   * Berechnet die Quadratwurzel einer Zahl mit Newtons Verfahren und hoher Genauigkeit.
3   * <p>
4   * Hinweis: Die maximale Anzahl an Nachkommastellen ist durch den Datentyp {@code int} für den
5   * Bei der aktuellen Implementierung mit {@code maxIterations = Math.max(100, scale * 5)} könne
6   * zuverlässig berechnet werden. Wird ein höherer Wert für {@code scale} angegeben, kann eine {
7   * </p>
8   * @param n Die Zahl, deren Quadratwurzel berechnet werden soll (n ≥ 0)
9   * @param scale Anzahl der gewünschten Nachkommastellen (maximal ca. 429.496.729)
10  * @return Die berechnete Quadratwurzel von n als BigDecimal
11  * @throws ArithmeticException falls die maximale Iterationszahl überschritten wird und das Erg
12  */
```

Fazit:

Die Methode ist für bis zu ca. 429 Millionen Nachkommastellen präzise, sofern genügend Speicher und Rechenzeit vorhanden sind. Für noch höhere Präzision müsste der Iterationszähler auf `long` oder einen größeren Datentyp umgestellt werden.

Beispielhafte Verwendung von GPT-4.1

7. Schritt: Kritisches Hinterfragen des Codes

User

Wenn ich in die Methode `Int.MAX_VALUE` reingebe, dann ist die Anzahl der Iterationen nicht wie erwartet. Passe den Code so an, dass bei einem zu großen Wert von `scale` eine `Exception` geworfen wird und passe auch die Dokumentation entsprechend an.

Agent

Die Methode wurde angepasst, sodass bei einem zu großen Wert für `scale` eine `IllegalArgumentException` geworfen wird. Die Dokumentation wurde ebenfalls entsprechend erweitert.

Hier ist der aktualisierte Ausschnitt:

```
1  /**
2   * Berechnet die Quadratwurzel einer Zahl mit Newtons Verfahren und hoher Genauigkeit.
3   * <p>
4   * Hinweis: Die maximale Anzahl an Nachkommastellen ist durch den Datentyp {@code int} für den
5   * Bei der aktuellen Implementierung mit {@code maxIterations = Math.max(100, scale * 5)} könne
6   * zuverlässig berechnet werden. Wird ein höherer Wert für {@code scale} angegeben, wird eine {
7   * </p>
8   * @param n Die Zahl, deren Quadratwurzel berechnet werden soll ( $n \geq 0$ )
9   * @param scale Anzahl der gewünschten Nachkommastellen (maximal ca. 429496729)
10  * @return Die berechnete Quadratwurzel von n als BigDecimal
11  * @throws IllegalArgumentException falls {@code scale} zu groß ist oder {@code n} negativ ist
12  * @throws ArithmeticException falls die maximale Iterationszahl überschritten wird und das Erg
13  */
14  public static BigDecimal sqrt(BigDecimal n, int scale) {
15      if (n.compareTo(BigDecimal.ZERO) < 0) {
16          throw new IllegalArgumentException("n muss  $\geq 0$  sein");
17      }
18      if (n.compareTo(BigDecimal.ZERO) == 0) return BigDecimal.ZERO;
19
20      // Begrenzung für scale, damit maxIterations nicht über Integer.MAX_VALUE hinausgeht
21      int maxScale = Integer.MAX_VALUE / 5;
22      if (scale > maxScale) {
23          throw new IllegalArgumentException(
24              "scale ist zu groß. Maximal erlaubt: " + maxScale
25          );
26      }
27
28      BigDecimal two = BigDecimal.valueOf(2);
29      MathContext mc = new MathContext(scale + 5, RoundingMode.HALF_UP); // Extra Stellen für Gen
30      BigDecimal x = n;
31      BigDecimal lastX;
32      int maxIterations = Math.max(100, scale * 5); // Dynamisch abhängig von scale
33      int count = 0;
34
35      do {
36          lastX = x;
37          x = n.divide(x, mc).add(x).divide(two, mc);
38          count++;
39      } while (
40          x.subtract(lastX).abs().compareTo(BigDecimal.ONE.scaleByPowerOfTen(-scale)) >
41              0 &&
42          count < maxIterations
```

```

43     );
44
45     if (count >= maxIterations) {
46         throw new ArithmeticException(
47             "Maximale Iterationszahl (" +
48             maxIterations +
49             ") erreicht, Ergebnis möglicherweise ungenau."
50         );
51     }
52
53     return x.setScale(scale, RoundingMode.HALF_UP);
54 }

```

Damit ist die Methode jetzt robust gegen zu große Werte für `scale` und die Dokumentation informiert klar über die Begrenzung.

Beispielhafte Verwendung von GPT-4.1

8. Schritt: Etwas Fordern, dass nicht bzw. nur begrenzt sinnvoll ist

Erzeuge mir ein einfaches npm Build-Script, dass Prettier verwendet, um den Code zu formatieren. Der Code soll eine maximale Breite von 70 Zeichen haben.

Hier ist ein einfaches npm Build-Script, das Prettier verwendet, um den Code zu formatieren. Die maximale Zeilenbreite ist auf 70 Zeichen gesetzt.

1. *package.json*

Das Format-Script und die Prettier-Dependency sind enthalten:

```
{
  "name": "sqr-jshell-vibecoded",
  "version": "1.0.0",
  "description": "Build-Script für Prettier-Codeformatierung mit max. 70 Zeichen Breite",
  "scripts": {
    "format": "prettier --write ."
  },
  "devDependencies": {
    "prettier": "^3.0.0"
  },
  "prettier": {
    "printWidth": 70
  }
}
```

2. *Prettier-Konfiguration (.prettierrc)*

Die maximale Breite ist explizit gesetzt:

```
{
  "printWidth": 70
}
```

Verwendung:

1. Installiere die Dependencies: `sh npm install`
2. Formatiere den Code: `sh npm run format`

Damit wird dein Code mit Prettier und einer Zeilenbreite von 70 Zeichen formatiert.

Warnung

Im Kontext von Java ist die Verwendung von Maven oder Gradle zum Bauen von Projekten der Standard. Es gab hier keinen Hinweis darauf, dass dies zumindest ungewöhnlich ist.

Weitere Prompts, um das Build-Skript zu vervollständigen in Hinblick auf Testausführung und Berechnung der Codeabdeckung haben erst viele Nachbesserungen erfordert und waren am Ende auch nicht vollständig erfolgreich.

Davon unabhängig ist auch die obige Konfiguration schon fragwürdig, da die Einstellung der maximalen Zeilenlänge an zwei Stellen erfolgt ist und diese Konfiguration so nicht funktionsfähig ist, da die Java Dateien, die zu formatieren sind, nicht angegeben wurden.

Weiterhin ist Prettier ungeeignet, da es keine Dokumentation umbrechen kann.

Beispielhafte Verwendung von GPT-4.1

9. Schritt: Tests generieren lassen

User

Bitte generiere mir Tests für sqrt Methode.

Agent

Hier findest du eine vollständige JUnit-Testklasse für die ``sqrt``-Methode. Die Datei liegt unter `./test/BigDecimalNewtonSqrtTest.java`.

Die Tests prüfen:

- Standardfälle (0, 1, 4, 9, 2 mit hoher Präzision)
- Negative Eingabe
- Zu große Präzision (`scale`)
- Sehr große und sehr kleine Zahlen

```
1  import org.junit.jupiter.api.Test;
2  import org.junit.jupiter.api.DisplayName;
3  import org.junit.jupiter.api.function.Executable;
4
5  import java.math.BigDecimal;
6
7  import static org.junit.jupiter.api.Assertions.*;
8
9  class BigDecimalNewtonSqrtTest {
10
11     @Test
12     @DisplayName("Quadratwurzel von 0 ist 0")
13     void testSqrtZero() {
14         BigDecimal result = BigDecimalNewtonSqrt.sqrt(BigDecimal.ZERO, 10);
15         assertEquals(BigDecimal.ZERO.setScale(10), result);
16     }
17
18     @Test
19     @DisplayName("Quadratwurzel von 1 ist 1")
20     void testSqrtOne() {
21         BigDecimal result = BigDecimalNewtonSqrt.sqrt(BigDecimal.ONE, 10);
22         assertEquals(BigDecimal.ONE.setScale(10), result);
23     }
24
25     @Test
26     @DisplayName("Quadratwurzel von 4 ist 2")
27     void testSqrtFour() {
28         BigDecimal result = BigDecimalNewtonSqrt.sqrt(new BigDecimal("4"), 10);
29         assertEquals(new BigDecimal("2.0000000000"), result);
30     }
31
32     @Test
33     @DisplayName("Quadratwurzel von 2 mit 20 Nachkommastellen")
34     void testSqrtTwoHighPrecision() {
35         BigDecimal expected = new BigDecimal("1.41421356237309504880");
36         BigDecimal result = BigDecimalNewtonSqrt.sqrt(new BigDecimal("2"), 20);
37         // Vergleiche die ersten 20 Nachkommastellen
38         assertEquals(expected, result);
39     }
40
41     @Test
42     @DisplayName("Quadratwurzel von 9 ist 3")
43     void testSqrtNine() {
44         BigDecimal result = BigDecimalNewtonSqrt.sqrt(new BigDecimal("9"), 8);
45         assertEquals(new BigDecimal("3.00000000"), result);
46     }
47 }
```

```

47
48 @Test
49 @DisplayName("Negative Eingabe wirft IllegalArgumentException")
50 void testNegativeInputThrows() {
51     assertThrows(IllegalArgumentException.class, () ->
52         BigDecimalNewtonSqrt.sqrt(new BigDecimal("-1"), 10)
53     );
54 }
55
56 @Test
57 @DisplayName("Zu großer scale wirft IllegalArgumentException")
58 void testTooLargeScaleThrows() {
59     int tooLargeScale = Integer.MAX_VALUE / 5 + 1;
60     assertThrows(IllegalArgumentException.class, () ->
61         BigDecimalNewtonSqrt.sqrt(BigDecimal.ONE, tooLargeScale)
62     );
63 }
64
65 @Test
66 @DisplayName("Quadratwurzel von sehr großer Zahl")
67 void testSqrtLargeNumber() {
68     BigDecimal large = new BigDecimal("1000000000000");
69     BigDecimal result = BigDecimalNewtonSqrt.sqrt(large, 5);
70     assertEquals(new BigDecimal("1000000.00000"), result);
71 }
72
73 @Test
74 @DisplayName("Quadratwurzel von kleiner Zahl")
75 void testSqrtSmallNumber() {
76     BigDecimal small = new BigDecimal("0.0004");
77     BigDecimal result = BigDecimalNewtonSqrt.sqrt(small, 6);
78     assertEquals(new BigDecimal("0.020000"), result);
79 }
80 }

```

Du kannst die Tests mit JUnit 5 ausführen. Falls du Unterstützung für das Test-Setup brauchst, sag einfach Bescheid!

Beispielhafte Verwendung von GPT-4.1

10. Schritt: Umstieg auf Maven als Build-Tool

User

Bitte erstelle mir eine Maven Konfiguration. Wenn möglich soll die Maven Konfiguration auch die Code Formatierung übernehmen und auf ein Plug-in setzten, dass auch die Dokumentation formatieren kann.

Agent

Hier ist eine Maven-Konfiguration (pom.xml), die folgende Features bietet:

- **JUnit 5** für Tests
- **Spotless Maven Plugin** für die automatische Formatierung von Java-Code und JavaDoc-Kommentaren
- **Zeilenlänge** und **JavaDoc-Umbruch** auf 70 Zeichen

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>de.michael.sqrt</groupId>
  <artifactId>sqrt-jshell-vibecoded</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>

  <name>SQRT-JShell-VibeCoded</name>
  <description>Berechnung der Quadratwurzel mit Newtons Verfahren und Formatierung/Tests</description>

  <properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <junit.version>5.10.2</junit.version>
    <formatter.version>2.23.0</formatter.version>
  </properties>

  <dependencies>
    <!-- JUnit 5 Jupiter API & Engine -->
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>${junit.version}</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <!-- Maven Compiler Plugin -->
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.11.0</version>
        <configuration>
          <source>${maven.compiler.source}</source>
          <target>${maven.compiler.target}</target>
        </configuration>
      </plugin>

      <!-- Maven Surefire Plugin for running tests -->
```



```

<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.2.5</version>
  <configuration>
    <useSystemClassLoader>>false</useSystemClassLoader>
  </configuration>
</plugin>

<!-- Spotless Plugin for code and JavaDoc formatting -->
<plugin>
  <groupId>com.diffplug.spotless</groupId>
  <artifactId>spotless-maven-plugin</artifactId>
  <version>2.23.0</version>
  <executions>
    <execution>
      <goals>
        <goal>apply</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <java>
      <googleJavaFormat>
        <version>1.17.0</version>
        <style>AOSP</style>
      </googleJavaFormat>
      <target>src/**/*.java</target>
      <removeUnusedImports />
      <formatAnnotations />
      <importOrder>
        <order>java, javax, org, com</order>
      </importOrder>
      <licenseHeader>
        <!-- Optional: Add a license header if desired -->
      </licenseHeader>
      <endWithNewLine>true</endWithNewLine>
      <indent>
        <spaces>4</spaces>
      </indent>
      <trimTrailingWhitespace>true</trimTrailingWhitespace>
      <maxLength>70</maxLength>
      <javadocFormatter>
        <enabled>true</enabled>
        <wrapLength>70</wrapLength>
      </javadocFormatter>
    </java>
  </configuration>
</plugin>
</plugins>
</build>
</project>

```

Wichtige Hinweise:

- Die Formatierung und das JavaDoc-Wrapping übernimmt das Spotless-Plugin mit Google Java Format und aktiviertem JavaDoc-Formatter.

- Du kannst formatieren mit:

```
sh mvn spotless:apply
```

- Tests laufen wie gewohnt mit:

```
sh mvn test
```

Falls du weitere Anpassungen oder eine andere Formatierungsstrategie möchtest, gib einfach Bescheid!

Die Konfiguration war in der Form nicht korrekt konnte aber automatisch korrigiert werden.

User

```
Ich bekomme folgende Fehlermeldung: [ERROR] Failed to execute goal  
com.diffplug.spotless:spotless-maven-plugin:2.23.0:apply (default-cli) on project sqrt-  
jshell-vibecoded: Unable to parse configuration of mojocom.diffplug.spotless:spotless-  
maven-plugin:2.23.0:apply for parameter target: Cannot find 'target' in  
class com.diffplug.spotless.maven.java.Java
```

Beispielhafte Verwendung von GPT-4.1

10. Schritt: Verwendung aktueller Tools einfordern

User

Bitte aktualisiere die Versionen der Plug-ins auf die neuesten Versionen.

Agent

Alle Versionen in deiner `pom.xml` wurden auf die neuesten stabilen Versionen aktualisiert und als Properties hinterlegt, damit du sie zentral pflegen kannst. Hier sind die wichtigsten Änderungen:

- Java-Version: 21
- Spotless Maven Plugin: 2.45.0
- Google Java Format: 1.17.0
- Maven Compiler Plugin: 3.11.0
- Maven Surefire Plugin: 3.2.5
- JUnit Jupiter: 5.10.2

Achtung!

Keine einzige Version war halbwegs aktuell. Ein manuelles Eingreifen war erforderlich.

Aktuell sind/bzw waren zum Zeitpunkt der Erstellung der Folien die folgenden Versionen:

- Java-Version: 24
- Spotless Maven Plugin: 2.46.1
- Maven Compiler Plugin: 3.14.0
- Maven Surefire Plugin: 3.5.3
- JUnit Jupiter: 5.13.4

Bemerkung

Ein expliziter Prompt nach neuen Versionen im Internet zu suchen führte zu folgendem Fehler:

Agent

```
Failed to connect to API: 400 Bad Request {"error":{"message":"prompt token count of 154533 exceeds the limit of 128000","code":"model_max_prompt_tokens_exceeded"}}
```

[D.h. das Ergebnisse der Webabfrage durch den Agenten ist zu groß und konnte von diesem nicht verarbeitet werden.]

Beispielhafte Verwendung von GPT-4.1

11. Schritt: Build-Skript korrigieren

User

Wenn ich mvn test ausführe, dann bekomme ich: "No tests to run." Bitte behebe das Problem.

Agent

Das Problem lag daran, dass sich deine Testklasse nicht im Standard-Maven-Testverzeichnis (`src/test/java`) befand. Ich habe die Datei jetzt nach `src/test/java/BigDecimalNewtonSqrtTest.java` verschoben.

Jetzt erkennt Maven die Tests automatisch.

Du kannst die Tests wie gewohnt mit

```
mvn test
```

ausführen.

Falls du weitere Testklassen hast, sollten diese ebenfalls im Verzeichnis `src/test/java` liegen und das Package-Statement (falls verwendet) korrekt gesetzt sein. Wenn du noch ein Package hinzufügen möchtest, kann ich das gerne für dich erledigen!

Achtung!

An diese Stelle wurde zwar der Test richtig verschoben, aber der eigentliche Code nicht. Es waren in der Folge mehrere Prompts, mit den entsprechenden Fehlermeldungen notwendig, damit die Projektstruktur angepasst wurde. Hier wäre eine manuelle Korrektur schneller gewesen. Erst danach konnten die Tests ausgeführt werden.

Beispielhafte Verwendung von GPT-4.1

12. Schritt: Fehler korrigieren

User

Ein Test schlägt fehl: [ERROR] BigDecimalNewtonSqrtTest.testSqrtZero – Time elapsed: 0.003 s <<< FAILURE! org.opentest4j.AssertionFailedError: expected: <0E-10> but was: <0>

Agent

Das Problem entsteht, weil `BigDecimal.ZERO.setScale(10)` ergibt `0E-10`, während das Ergebnis der Methode `sqr`t einfach `0` ist (Skalierung 0). Obwohl beide Werte mathematisch gleich sind, unterscheidet `BigDecimal.equals()` zwischen der Skala.

...

Achtung!

Auch hier hat es mehrere Runden gedauert bis die Tests einwandfrei funktioniert haben. Der dabei entstandene Testcode ist jedoch klar Verbesserungsfähig, da er auf Stringvergleich (zum Beispiel: `:java: assertEquals("0.0000000000", result.toPlainString());`) gesetzt hat, was nicht notwendig gewesen wäre. Fehler im Code gab es keine.

Beispielhafte Verwendung von GPT-4.1

13. Schritt: Code verbessern

User

Bitte ergänze den Code so, dass negative scale Werte sinnvoll behandelt werden.

Agent

Die Methode behandelt jetzt negative Werte für `scale` sinnvoll:

```
1 | if (scale < 0) {  
2 |     throw new IllegalArgumentException("scale muss ≥ 0 sein");  
3 | }
```

Das bedeutet:

Wenn ein negativer Wert für `scale` übergeben wird, wird eine aussagekräftige `IllegalArgumentException` geworfen. Damit ist die Eingabeprüfung für die Präzision jetzt vollständig und robust.

Achtung!

Der Testcode wurde nicht mit bedacht!

Beispielhafte Verwendung von GPT-4.1

14. Schritt: Tests nachziehen

Bitte ergänze auch die Tests.

User

Die Tests wurden ergänzt:

Es gibt jetzt einen Test, der prüft, dass ein negativer Wert für `scale` eine `IllegalArgumentException` auslöst:

```
1  @Test
2  @DisplayName("Negative scale wirft IllegalArgumentException")
3  void testNegativeScaleThrows() {
4      assertThrows(IllegalArgumentException.class, () ->
5          BigDecimalNewtonSqrt.sqrt(BigDecimal.ONE, -1)
6      );
7  }
```

Damit ist die Eingabeprüfung für negative `scale`-Werte auch durch die Tests abgedeckt.

Agent

Beispielhafte Verwendung von GPT-4.1

15. Schritt: Build-Skript vervollständigen

User

Bitte ergänze das Build-Skript so, dass die Testabdeckung berechnet wird.

Agent

Um die Testabdeckung in einem Maven-Projekt zu berechnen, wird üblicherweise das Plugin **JaCoCo** verwendet. Dieses Plugin erstellt Berichte über die Testabdeckung (Coverage) und lässt sich einfach in die `pom.xml` integrieren.

Hier ist die Ergänzung für dein Build-Skript:

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.11</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>test</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Achtung!

Auch hier war eine manuelle Anpassung der Version erforderlich. Die angegebene Version war mit dem Projekt nicht kompatibel.

Beispielhafte Verwendung von GPT-4.1

16. Schritt: Basierend auf Codeabdeckung weitere Tests fordern

User

Die Testabdeckung sagt, dass die Zeile 62 noch nicht getestet wird. Kannst Du dafür einen Testfall generieren?

Agent

Die Zeile 62 ist:

```
1 if (count >= maxIterations) {  
2     throw new ArithmeticException(  
3         "Maximale Iterationszahl (" +  
4         maxIterations +  
5         ") erreicht, Ergebnis möglicherweise ungenau."  
6     );  
7 }
```

Das bedeutet:

Du brauchst einen Test, der absichtlich dafür sorgt, dass die maximale Iterationszahl überschritten wird.

Wie erreichst du das?

- Wähle einen Wert für `n`, der schwer konvergiert (z.B. eine sehr große Zahl mit sehr hoher Präzision).
- Setze `scale` auf einen Wert > 0 , aber so hoch, dass die Konvergenz nicht innerhalb von `maxIterations` erreicht wird.
- Oder: Manipuliere die Methode für den Test, sodass sie nie konvergiert (das ist aber unsauber).

Praktischer Testfall:

Setze `scale` auf einen sehr hohen Wert (z.B. `Integer.MAX_VALUE / 5`), aber das wirft schon vorher eine `IllegalArgumentException`. Deshalb: Wähle einen Wert, bei dem die Methode zwar startet, aber nicht konvergiert. Ein Ansatz ist, eine Zahl zu wählen, die sehr klein ist und eine sehr hohe Präzision verlangt.

Achtung!

Hier war auch nach mehreren Versuchen es nicht möglich einen Testfall zu finden, der die maximale Iterationszahl erreicht.

Beispielhafte Verwendung von GPT-4.1

Zusammenfassung

- Gesamtaufwand ca. 2 Stunden
- An mehreren Stellen war manuelles Eingreifen erforderlich
- Man kann Dinge einfordern, die wenig Sinn machen, ohne dass man (unmittelbar) darauf aufmerksam gemacht wird (z. B. Verwendung von npm)
- An vielen Stellen war das einfache Kopieren von Fehlermeldungen in den Chat ausreichend, um die Fehler korrigiert zu bekommen.
- Wenn der Prompt nicht präzise genug ist, dann muss man (ggf. mehrfach) prompten.
- Kritisches Hinterfragen des Codes ist dringend notwendig. Nur dann erhält man Code, der auch höhere Qualitätsstandards erreicht.
- Häufig wird die nicht die Codebasis als Ganzes betrachtet. Zum Beispiel brauchen ggf. Anpassungen von Dokumenten, Skripts und Tests nachfolgende Prompts.

Beispielhafte Verwendung von Claude Sonnet 4 Thinking

Das Ziel ist (auch hier) die Generierung einer Kommandozeilenanwendung zur Berechnung der Quadratwurzel einer Zahl mit vorgegebener, aber beliebiger Genauigkeit.

Der Unterschied ist jedoch, dass wir versuchen das ganze Projekt in einem Zug generieren zu lassen, in dem wir einen Prompt erstellen, der „alle“ Anforderungen des Projekts beschreibt.

Beispielhafte Verwendung von Claude Sonnet 4 Thinking

Initialer Prompt

User

Bitte generiere mir ein Java-Kommandozeilenprogramm zur Berechnung der Quadratwurzel unter Verwendung der Newtown-Raphson-Methode. Die Anforderungen sind:

- Es soll möglich sein die gewünschte Anzahl an Nachkommastellen anzugeben.
- Die Methode zur Berechnung der Quadratwurzel soll auch aus anderem Code heraus aufrufbar sein.
- Die Methode soll robust sein bei fehlerhaften Eingaben und ggf. aussagekräftige Ausnahmen werfen.
- Die Methode zur Berechnung der Quadratwurzel soll über eine umfassende Dokumentation verfügen, die auch alle Sonderfälle und Ausnahmen dokumentiert.
- Es soll nach Möglichkeit nur eine Datei generiert werden, um eine optionale Verwendung als Skript einfach zu ermöglichen.
- Das Programm soll die neueste Version von Java verwenden.
- Alle verwendeten Bibliotheken und Tools sollen auf dem neuesten Stand sein.
- Es soll ein Build-Skript geben, dass den Code formatiert und testet. Bei der Codeformatierung soll insbesondere eine maximale Zeilenlänge von 70 Zeichen eingehalten werden. Die Tests sollen die Methode, die die Berechnung durchführt, umfassend testen; inkl. Sonderfälle. Es soll auch eine Codeabdeckung berechnet werden und nach Möglichkeit 100% Branch-Coverage erreicht werden.

Beispielhafte Verwendung von Claude Sonnet 4 Thinking

(🇬🇧 *Thought Process*)

<think>

Das ist eine umfassende Anforderung für ein Java-Kommandozeilenprogramm. Lass mich die Anforderungen durchgehen:

1. Java-Kommandozeilenprogramm zur Berechnung der Quadratwurzel mit Newton-Raphson-Methode
2. Anzahl der Nachkommastellen soll konfigurierbar sein
3. Methode soll aus anderem Code aufrufbar sein
4. Robust bei fehlerhaften Eingaben mit aussagekräftigen Ausnahmen
5. Umfassende Dokumentation inkl. Sonderfälle
6. Nur eine Datei wenn möglich (für Skript-Verwendung)
7. Neueste Java-Version
8. Neueste Bibliotheken
9. Build-Skript mit Formatierung (max 70 Zeichen Zeilenlänge) und Tests
10. Umfassende Tests inkl. Sonderfälle
11. 100% Branch-Coverage angestrebt

Ich sollte zuerst schauen, was im aktuellen Projektverzeichnis ist, und dann eine Struktur aufbauen.

Newton-Raphson für Quadratwurzel: $x_{n+1} = (x_n + a/x_n)/2$ wobei a die Zahl ist, deren Quadratwurzel gesucht wird.

Ich werde:

1. Eine Java-Klasse erstellen mit der Newton-Raphson Implementierung
2. Kommandozeilenhandling hinzufügen
3. Robuste Fehlerbehandlung
4. Umfassende Dokumentation
5. Ein Build-Skript (build.gradle oder Maven) für Formatierung und Tests
6. JUnit-Tests mit hoher Abdeckung

Für maximale Flexibilität werde ich eine einzelne Java-Datei entwickeln, die sowohl als Skript als auch als importierbare Bibliothek funktioniert. Das bedeutet, ich integriere Kommandozeilen-Logik direkt in die Hauptklasse, sodass sie sowohl standalone als auch als Komponente in anderen Projekten nutzbar ist.

</think>

Bemerkung

Danach erfolgt ein langwieriger Prozess in dem das Model die einzelnen Schritte abarbeitet und dabei immer wieder überprüft ob das erwartete Ergebnis erreicht wurde. Stellt es dabei Probleme fest, zum Beispiel beim Ausführen des Build-Skriptes oder der Tests, dann versucht es selber diese zu korrigieren. Dies ist im Chat genau nachlesbar.

Beispielhafte Verwendung von Claude Sonnet 4 Thinking

Generierter Code

```
1  import java.math.BigDecimal;
2  import java.math.MathContext;
3  import java.math.RoundingMode;
4  import java.util.Scanner;
5
6  /**
7   * Ein hochpräziser Quadratwurzel-Rechner basierend auf der
8   * Newton-Raphson-Methode.
9   *
10  * Diese Klasse bietet sowohl eine Kommandozeilenschnittstelle als auch
11  * eine programmierbare API zur Berechnung von Quadratwurzeln mit
12  * beliebiger Genauigkeit.
13  *
14  * @author Generated Code
15  * @version 1.0
16  * @since Java 21
17  */
18  public class SquareRootCalculator {
19
20      /**
21       * Standard-Präzision für Berechnungen (50 Dezimalstellen).
22       */
23      private static final int DEFAULT_PRECISION = 50;
24
25      /**
26       * Maximale erlaubte Präzision zur Vermeidung von
27       * Performance-Problemen.
28       */
29      private static final int MAX_PRECISION = 1000;
30
31      /**
32       * Maximale Anzahl von Iterationen für die Newton-Raphson-Methode.
33       */
34      private static final int MAX_ITERATIONS = 10000;
35
36      /**
37       * Custom Exception für ungültige Eingaben.
38       */
39      public static class InvalidInputException extends Exception {
40          public InvalidInputException(String message) {
41              super(message);
42          }
43      }
44
45      /**
46       * Custom Exception für Berechnungsfehler.
47       */
48      public static class CalculationException extends Exception {
49          public CalculationException(String message) {
50              super(message);
51          }
52      }
53
54      /**
55       * Berechnet die Quadratwurzel einer Zahl mit der
56       * Newton-Raphson-Methode.
57       *
58       * Die Newton-Raphson-Methode für Quadratwurzeln verwendet die
59       * Iterationsformel:
60       *  $x_{n+1} = (x_n + a/x_n) / 2$ 
61       *
```

```

62 * wobei 'a' die Eingabezahl und x_0 ein Startwert ist.
63 *
64 * @param number Die Zahl, deren Quadratwurzel berechnet werden soll.
65 *           Muss größer oder gleich 0 sein.
66 * @param decimalPlaces Die Anzahl der gewünschten Nachkommastellen.
67 *           Muss zwischen 0 und 1000 liegen.
68 * @return Die Quadratwurzel als BigDecimal mit der gewünschten
69 *         Präzision.
70 *
71 * @throws InvalidInputException wenn die Eingabe ungültig ist:
72 *     - number ist null
73 *     - number ist negativ
74 *     - decimalPlaces ist negativ oder größer als 1000
75 * @throws CalculationException wenn die Berechnung fehlschlägt:
76 *     - Konvergenz nicht erreicht nach MAX_ITERATIONS
77 *     - Numerischer Overflow/Underflow
78 *
79 * Sonderfälle:
80 * - sqrt(0) = 0
81 * - sqrt(1) = 1
82 * - Für sehr kleine Zahlen (< 1e-100) wird eine spezielle
83 *   Behandlung angewendet
84 * - Für sehr große Zahlen wird die Berechnung in Segmente
85 *   aufgeteilt
86 *
87 * Beispiele:
88 * <pre>
89 * { @code
90 * // Einfache Berechnung
91 * BigDecimal result = calculateSquareRoot(
92 *     new BigDecimal("2"), 5);
93 * // Ergebnis: 1.41421
94 *
95 * // Hochpräzise Berechnung
96 * BigDecimal precise = calculateSquareRoot(
97 *     new BigDecimal("2"), 50);
98 *
99 * // Große Zahlen
100 * BigDecimal large = calculateSquareRoot(
101 *     new BigDecimal("1234567890123456789"), 10);
102 * }
103 * </pre>
104 */
105 public static BigDecimal calculateSquareRoot(BigDecimal number,
106                                             int decimalPlaces)
107     throws InvalidInputException, CalculationException {
108
109     // Eingabevalidierung
110     if (number == null) {
111         throw new InvalidInputException(
112             "Die Eingabezahl darf nicht null sein.");
113     }
114
115     if (number.compareTo(BigDecimal.ZERO) < 0) {
116         throw new InvalidInputException(
117             "Die Quadratwurzel aus negativen Zahlen ist nicht " +
118             "im Bereich der reellen Zahlen definiert. " +
119             "Eingabe: " + number);
120     }
121
122     if (decimalPlaces < 0) {
123         throw new InvalidInputException(
124             "Die Anzahl der Nachkommastellen muss nicht-negativ " +
125             "sein. Eingabe: " + decimalPlaces);
126     }
127
128     if (decimalPlaces > MAX_PRECISION) {
129         throw new InvalidInputException(
130             "Die maximale Anzahl von Nachkommastellen beträgt " +

```

```

131         MAX_PRECISION + ". Eingabe: " + decimalPlaces);
132     }
133
134     // Sonderfall: sqrt(0) = 0
135     if (number.compareTo(BigDecimal.ZERO) == 0) {
136         return BigDecimal.ZERO.setScale(decimalPlaces,
137             RoundingMode.HALF_UP);
138     }
139
140     // Sonderfall: sqrt(1) = 1
141     if (number.compareTo(BigDecimal.ONE) == 0) {
142         return BigDecimal.ONE.setScale(decimalPlaces,
143             RoundingMode.HALF_UP);
144     }
145
146     // Präzision für interne Berechnungen (höher als gewünscht)
147     int internalPrecision = Math.max(decimalPlaces + 10,
148         DEFAULT_PRECISION);
149     MathContext mc = new MathContext(internalPrecision,
150         RoundingMode.HALF_UP);
151
152     try {
153         // Startwert bestimmen (heuristische Schätzung)
154         BigDecimal x = getInitialGuess(number, mc);
155         BigDecimal previousX;
156
157         // Newton-Raphson-Iteration
158         for (int i = 0; i < MAX_ITERATIONS; i++) {
159             previousX = x;
160
161             //  $x_{n+1} = (x_n + \text{number}/x_n) / 2$ 
162             BigDecimal quotient = number.divide(x, mc);
163             x = x.add(quotient).divide(BigDecimal.valueOf(2), mc);
164
165             // Konvergenzprüfung
166             if (hasConverged(x, previousX, decimalPlaces + 5)) {
167                 return x.setScale(decimalPlaces,
168                     RoundingMode.HALF_UP);
169             }
170         }
171
172         throw new CalculationException(
173             "Die Newton-Raphson-Methode konvergierte nicht nach " +
174             MAX_ITERATIONS + " Iterationen. Möglicherweise ist " +
175             "die gewünschte Präzision zu hoch oder die Eingabe " +
176             "numerisch instabil.");
177
178     } catch (ArithmeticException e) {
179         throw new CalculationException(
180             "Numerischer Fehler während der Berechnung: " +
181             e.getMessage() + ". Dies kann bei sehr großen oder " +
182             "sehr kleinen Zahlen auftreten.");
183     }
184 }
185
186 /**
187  * Bestimmt einen geeigneten Startwert für die Newton-Raphson-Iteration.
188  *
189  * @param number Die Zahl, deren Quadratwurzel berechnet werden soll
190  * @param mc Der MathContext für die Berechnung
191  * @return Ein geeigneter Startwert
192  */
193 private static BigDecimal getInitialGuess(BigDecimal number,
194     MathContext mc) {
195     // Für Zahlen nahe 1: verwende die Zahl selbst
196     if (number.compareTo(BigDecimal.valueOf(0.1)) >= 0 &&
197         number.compareTo(BigDecimal.valueOf(10)) <= 0) {
198         return number;
199     }

```



```

200
201 // Für sehr kleine Zahlen: verwende 1
202 if (number.compareTo(BigDecimal.valueOf(0.1)) < 0) {
203     return BigDecimal.ONE;
204 }
205
206 // Für große Zahlen: verwende number/2
207 return number.divide(BigDecimal.valueOf(2), mc);
208 }
209
210 /**
211  * Prüft, ob die Newton-Raphson-Iteration konvergiert ist.
212  *
213  * @param current Der aktuelle Wert
214  * @param previous Der vorherige Wert
215  * @param requiredPrecision Die erforderliche Präzision
216  * @return true, wenn konvergiert
217  */
218 private static boolean hasConverged(BigDecimal current,
219                                     BigDecimal previous,
220                                     int requiredPrecision) {
221     if (current.equals(previous)) {
222         return true;
223     }
224
225     BigDecimal difference = current.subtract(previous).abs();
226     BigDecimal tolerance = BigDecimal.ONE.divide(
227         BigDecimal.TEN.pow(requiredPrecision),
228         new MathContext(requiredPrecision + 10));
229
230     return difference.compareTo(tolerance) <= 0;
231 }
232
233 /**
234  * Überladene Methode mit Standard-Präzision von 10 Nachkommastellen.
235  *
236  * @param number Die Zahl, deren Quadratwurzel berechnet werden soll
237  * @return Die Quadratwurzel mit 10 Nachkommastellen
238  * @throws InvalidInputException bei ungültiger Eingabe
239  * @throws CalculationException bei Berechnungsfehlern
240  */
241 public static BigDecimal calculateSquareRoot(BigDecimal number)
242     throws InvalidInputException, CalculationException {
243     return calculateSquareRoot(number, 10);
244 }
245
246 /**
247  * Convenience-Methode für double-Werte.
248  *
249  * @param number Die Zahl als double
250  * @param decimalPlaces Anzahl Nachkommastellen
251  * @return Die Quadratwurzel
252  * @throws InvalidInputException bei ungültiger Eingabe
253  * @throws CalculationException bei Berechnungsfehlern
254  */
255 public static BigDecimal calculateSquareRoot(double number,
256                                             int decimalPlaces)
257     throws InvalidInputException, CalculationException {
258     if (Double.isNaN(number) || Double.isInfinite(number)) {
259         throw new InvalidInputException(
260             "Die Eingabe darf nicht NaN oder unendlich sein.");
261     }
262     return calculateSquareRoot(BigDecimal.valueOf(number),
263                               decimalPlaces);
264 }
265
266 /**
267  * Hauptmethode für die Kommandozeilenschnittstelle.
268  *

```

```

269 * Verwendung:
270 * - Ohne Argumente: Interaktiver Modus
271 * - Mit einem Argument: Berechnet sqrt(arg) mit 10 Nachkommastellen
272 * - Mit zwei Argumenten: Berechnet sqrt(arg1) mit arg2
273 * Nachkommastellen
274 *
275 * @param args Kommandozeilenargumente
276 */
277 public static void main(String[] args) {
278     try {
279         if (args.length == 0) {
280             runInteractiveMode();
281         } else if (args.length == 1) {
282             calculateAndPrint(args[0], "10");
283         } else if (args.length == 2) {
284             calculateAndPrint(args[0], args[1]);
285         } else {
286             printUsage();
287             System.exit(1);
288         }
289     } catch (Exception e) {
290         System.err.println("Fehler: " + e.getMessage());
291         System.exit(1);
292     }
293 }
294
295 /**
296  * Startet den interaktiven Modus.
297  */
298 private static void runInteractiveMode() {
299     Scanner scanner = new Scanner(System.in);
300
301     System.out.println("=====");
302     System.out.println("  Quadratwurzel-Rechner v1.0");
303     System.out.println("  Newton-Raphson-Methode");
304     System.out.println("=====");
305     System.out.println();
306
307     while (true) {
308         try {
309             System.out.print("Zahl (oder 'quit' zum Beenden): ");
310             String numberInput = scanner.nextLine().trim();
311
312             if ("quit".equalsIgnoreCase(numberInput) ||
313                 "exit".equalsIgnoreCase(numberInput)) {
314                 System.out.println("Auf Wiedersehen!");
315                 break;
316             }
317
318             System.out.print("Nachkommastellen (Standard: 10): ");
319             String precisionInput = scanner.nextLine().trim();
320
321             if (precisionInput.isEmpty()) {
322                 precisionInput = "10";
323             }
324
325             calculateAndPrint(numberInput, precisionInput);
326             System.out.println();
327
328         } catch (Exception e) {
329             System.err.println("Fehler: " + e.getMessage());
330             System.out.println();
331         }
332     }
333
334     scanner.close();
335 }
336

```

```

337  /**
338  * Berechnet und gibt das Ergebnis aus.
339  */
340  private static void calculateAndPrint(String numberStr,
341                                     String precisionStr)
342      throws InvalidInputException, CalculationException {
343
344      BigDecimal number;
345      int precision;
346
347      try {
348          number = new BigDecimal(numberStr);
349      } catch (NumberFormatException e) {
350          throw new InvalidInputException(
351              "" + numberStr + " ist keine gültige Zahl.");
352      }
353
354      try {
355          precision = Integer.parseInt(precisionStr);
356      } catch (NumberFormatException e) {
357          throw new InvalidInputException(
358              "" + precisionStr +
359              " ist keine gültige Anzahl von Nachkommastellen.");
360      }
361
362      long startTime = System.nanoTime();
363      BigDecimal result = calculateSquareRoot(number, precision);
364      long endTime = System.nanoTime();
365
366      System.out.println("Eingabe: " + number);
367      System.out.println("Nachkommastellen: " + precision);
368      System.out.println("Ergebnis: " + result);
369      System.out.printf("Berechnungszeit: %.2f ms%n",
370                      (endTime - startTime) / 1_000_000.0);
371
372      // Verifikation (Quadrat des Ergebnisses)
373      if (precision ≤ 20) { // Nur für kleinere Präzisionen
374          BigDecimal verification = result.multiply(result);
375          System.out.println("Verifikation (Quadrat): " +
376                          verification.setScale(precision,
377                          RoundingMode.HALF_UP));
378      }
379  }
380
381  /**
382  * Gibt die Verwendungsinformationen aus.
383  */
384  private static void printUsage() {
385      System.out.println("Quadratwurzel-Rechner - Newton-Raphson-Methode");
386      System.out.println();
387      System.out.println("Verwendung:");
388      System.out.println("  java SquareRootCalculator");
389      System.out.println("  Startet den interaktiven Modus");
390      System.out.println();
391      System.out.println("  java SquareRootCalculator <zahl>");
392      System.out.println("  Berechnet sqrt(<zahl>) mit 10 " +
393          "Nachkommastellen");
394      System.out.println();
395      System.out.println("  java SquareRootCalculator <zahl> " +
396          "<nachkommastellen>");
397      System.out.println("  Berechnet sqrt(<zahl>) mit der " +
398          "angegebenen Präzision");
399      System.out.println();
400      System.out.println("Beispiele:");
401      System.out.println("  java SquareRootCalculator 2");
402      System.out.println("  java SquareRootCalculator 2 20");
403      System.out.println("  java SquareRootCalculator 1234567890.123456");
404  }
405  }

```


Beispielhafte Verwendung von Claude Sonnet 4 Thinking

Generierte Tests

```
1 import static org.assertj.core.api.Assertions.*;
2 import static org.junit.jupiter.api.Assertions.*;
3
4 import java.io.ByteArrayInputStream;
5 import java.io.ByteArrayOutputStream;
6 import java.io.PrintStream;
7 import java.math.BigDecimal;
8 import java.math.RoundingMode;
9 import java.util.stream.Stream;
10 import org.junit.jupiter.api.AfterEach;
11 import org.junit.jupiter.api.BeforeEach;
12 import org.junit.jupiter.api.DisplayName;
13 import org.junit.jupiter.api.Test;
14 import org.junit.jupiter.api.TestInstance;
15 import org.junit.jupiter.params.ParameterizedTest;
16 import org.junit.jupiter.params.provider.Arguments;
17 import org.junit.jupiter.params.provider.MethodSource;
18 import org.junit.jupiter.params.provider.ValueSource;
19
20 @TestInstance(TestInstance.Lifecycle.PER_CLASS)
21 @DisplayName("SquareRootCalculator Tests")
22 class SquareRootCalculatorTest {
23
24     private final ByteArrayOutputStream outputStreamCaptor =
25         new ByteArrayOutputStream();
26     private final ByteArrayOutputStream errorStreamCaptor =
27         new ByteArrayOutputStream();
28     private final PrintStream standardOut = System.out;
29     private final PrintStream standardErr = System.err;
30
31     @BeforeEach
32     void setUp() {
33         System.setOut(new PrintStream(outputStreamCaptor));
34         System.setErr(new PrintStream(errorStreamCaptor));
35     }
36
37     @AfterEach
38     void tearDown() {
39         System.setOut(standardOut);
40         System.setErr(standardErr);
41     }
42
43     @Test
44     @DisplayName("Should calculate square root of 2 with default precision")
45     void testSquareRootOfTwoDefaultPrecision() throws Exception {
46         BigDecimal result = SquareRootCalculator.calculateSquareRoot(
47             new BigDecimal("2"));
48
49         assertThat(result).isNotNull();
50         assertThat(result.scale()).isEqualTo(10);
51
52         // Verify precision (should be close to 1.4142135623)
53         BigDecimal expected = new BigDecimal("1.4142135624");
54         assertThat(result.subtract(expected).abs())
55             .isLessThan(new BigDecimal("0.0000000001"));
56     }
57
58     @Test
59     @DisplayName("Should calculate square root with specified precision")
60     void testSquareRootWithSpecifiedPrecision() throws Exception {
61         BigDecimal result = SquareRootCalculator.calculateSquareRoot(
62             new BigDecimal("2"), 5);
63     }
64 }
```

```

63         assertThat(result.scale()).isEqualTo(5);
64         assertThat(result.toString()).isEqualTo("1.41421");
65     }
66
67
68     @Test
69     @DisplayName("Should return 0 for square root of 0")
70     void testSquareRootOfZero() throws Exception {
71         BigDecimal result = SquareRootCalculator.calculateSquareRoot(
72             BigDecimal.ZERO, 10);
73
74         assertThat(result).isEqualTo(
75             BigDecimal.ZERO.setScale(10, RoundingMode.HALF_UP));
76     }
77
78     @Test
79     @DisplayName("Should return 1 for square root of 1")
80     void testSquareRootOfOne() throws Exception {
81         BigDecimal result = SquareRootCalculator.calculateSquareRoot(
82             BigDecimal.ONE, 10);
83
84         assertThat(result).isEqualTo(
85             BigDecimal.ONE.setScale(10, RoundingMode.HALF_UP));
86     }
87
88     @ParameterizedTest
89     @MethodSource("validSquareRootTestCases")
90     @DisplayName("Should calculate square roots for various valid inputs")
91     void testValidSquareRoots(BigDecimal input, int precision,
92         BigDecimal expectedApprox) throws Exception {
93         BigDecimal result = SquareRootCalculator.calculateSquareRoot(
94             input, precision);
95
96         assertThat(result).isNotNull();
97         assertThat(result.scale()).isEqualTo(precision);
98
99         // Verify by squaring the result
100        BigDecimal squared = result.multiply(result);
101        BigDecimal tolerance = BigDecimal.ONE.divide(
102            BigDecimal.TEN.pow(Math.min(precision, 10)));
103
104        assertThat(squared.subtract(input).abs())
105            .isLessThanOrEqualTo(tolerance);
106    }
107
108    private static Stream<Arguments> validSquareRootTestCases() {
109        return Stream.of(
110            Arguments.of(new BigDecimal("4"), 5, new BigDecimal("2")),
111            Arguments.of(new BigDecimal("9"), 3, new BigDecimal("3")),
112            Arguments.of(new BigDecimal("16"), 2, new BigDecimal("4")),
113            Arguments.of(new BigDecimal("0.25"), 8, new BigDecimal("0.5")),
114            Arguments.of(new BigDecimal("0.01"), 10,
115                new BigDecimal("0.1")),
116            Arguments.of(new BigDecimal("100"), 1, new BigDecimal("10")),
117            Arguments.of(new BigDecimal("1000000"), 15,
118                new BigDecimal("1000")),
119            Arguments.of(new BigDecimal("0.000001"), 20,
120                new BigDecimal("0.001"))
121        );
122    }
123
124    @Test
125    @DisplayName("Should throw InvalidInputException for null input")
126    void testNullInput() {
127        SquareRootCalculator.InvalidInputException exception =
128            assertThrows(SquareRootCalculator.InvalidInputException.class,
129                () -> SquareRootCalculator.calculateSquareRoot(
130                    null, 10));
131    }

```

```

132         assertThat(exception.getMessage())
133             .contains("darf nicht null sein");
134     }
135
136     @Test
137     @DisplayName("Should throw InvalidInputException for negative input")
138     void testNegativeInput() {
139         SquareRootCalculator.InvalidInputException exception =
140             assertThrows(SquareRootCalculator.InvalidInputException.class,
141                 () -> SquareRootCalculator.calculateSquareRoot(
142                     new BigDecimal("-1"), 10));
143
144         assertThat(exception.getMessage())
145             .contains("negativen Zahlen")
146             .contains("-1");
147     }
148
149     @Test
150     @DisplayName("Should throw InvalidInputException for negative decimal places")
151     void testNegativeDecimalPlaces() {
152         SquareRootCalculator.InvalidInputException exception =
153             assertThrows(SquareRootCalculator.InvalidInputException.class,
154                 () -> SquareRootCalculator.calculateSquareRoot(
155                     new BigDecimal("2"), -1));
156
157         assertThat(exception.getMessage())
158             .contains("nicht-negativ")
159             .contains("-1");
160     }
161
162     @Test
163     @DisplayName("Should throw InvalidInputException for too high precision")
164     void testTooHighPrecision() {
165         SquareRootCalculator.InvalidInputException exception =
166             assertThrows(SquareRootCalculator.InvalidInputException.class,
167                 () -> SquareRootCalculator.calculateSquareRoot(
168                     new BigDecimal("2"), 1001));
169
170         assertThat(exception.getMessage())
171             .contains("maximale Anzahl")
172             .contains("1000")
173             .contains("1001");
174     }
175
176     @Test
177     @DisplayName("Should handle maximum allowed precision")
178     void testMaximumPrecision() throws Exception {
179         BigDecimal result = SquareRootCalculator.calculateSquareRoot(
180             new BigDecimal("2"), 1000);
181
182         assertThat(result).isNotNull();
183         assertThat(result.scale()).isEqualTo(1000);
184     }
185
186     @Test
187     @DisplayName("Should calculate square root for double input")
188     void testDoubleInput() throws Exception {
189         BigDecimal result = SquareRootCalculator.calculateSquareRoot(
190             4.0, 5);
191
192         assertThat(result.toString()).isEqualTo("2.00000");
193     }
194
195     @Test
196     @DisplayName("Should throw InvalidInputException for NaN double input")
197     void testNaNDoubleInput() {
198         SquareRootCalculator.InvalidInputException exception =
199             assertThrows(SquareRootCalculator.InvalidInputException.class,
200                 () -> SquareRootCalculator.calculateSquareRoot(

```



```

201         Double.NaN, 5));
202
203         assertThat(exception.getMessage())
204             .contains("NaN oder unendlich");
205     }
206
207     @Test
208     @DisplayName("Should throw InvalidInputException for infinite double input")
209     void testInfiniteDoubleInput() {
210         SquareRootCalculator.InvalidInputException exception =
211             assertThrows(SquareRootCalculator.InvalidInputException.class,
212                 () -> SquareRootCalculator.calculateSquareRoot(
213                     Double.POSITIVE_INFINITY, 5));
214
215         assertThat(exception.getMessage())
216             .contains("NaN oder unendlich");
217     }
218
219     @Test
220     @DisplayName("Should handle very small numbers")
221     void testVerySmallNumbers() throws Exception {
222         BigDecimal result = SquareRootCalculator.calculateSquareRoot(
223             new BigDecimal("1e-50"), 10);
224
225         assertThat(result).isNotNull();
226         assertThat(result.scale()).isEqualTo(10);
227     }
228
229     @Test
230     @DisplayName("Should handle very large numbers")
231     void testVeryLargeNumbers() throws Exception {
232         BigDecimal result = SquareRootCalculator.calculateSquareRoot(
233             new BigDecimal("1e50"), 10);
234
235         assertThat(result).isNotNull();
236         assertThat(result.scale()).isEqualTo(10);
237     }
238
239     @Test
240     @DisplayName("Should handle numbers close to 1")
241     void testNumbersCloseToOne() throws Exception {
242         BigDecimal result = SquareRootCalculator.calculateSquareRoot(
243             new BigDecimal("1.000001"), 15);
244
245         assertThat(result).isNotNull();
246         assertThat(result.scale()).isEqualTo(15);
247     }
248
249     @Test
250     @DisplayName("Should run main with no arguments (interactive mode mock)")
251     void testMainWithNoArguments() throws Exception {
252         // Mock interactive input
253         String input = "4\n5\nquit\n";
254         System.setIn(new ByteArrayInputStream(input.getBytes()));
255
256         SquareRootCalculator.main(new String[]{});
257
258         String output = outputStreamCaptor.toString();
259         assertThat(output).contains("Quadratwurzel-Rechner v1.0");
260         assertThat(output).contains("Eingabe: 4");
261         assertThat(output).contains("Ergebnis: 2.00000");
262         assertThat(output).contains("Auf Wiedersehen!");
263     }
264
265     @Test
266     @DisplayName("Should run main with one argument")
267     void testMainWithOneArgument() throws Exception {
268         SquareRootCalculator.main(new String[]{"9"});
269     }

```



```

270     String output = outputStreamCaptor.toString();
271     assertThat(output).contains("Eingabe: 9");
272     assertThat(output).contains("Nachkommastellen: 10");
273     assertThat(output).contains("Ergebnis: 3.0000000000");
274 }
275
276 @Test
277 @DisplayName("Should run main with two arguments")
278 void testMainWithTwoArguments() throws Exception {
279     SquareRootCalculator.main(new String[]{"16", "3"});
280
281     String output = outputStreamCaptor.toString();
282     assertThat(output).contains("Eingabe: 16");
283     assertThat(output).contains("Nachkommastellen: 3");
284     assertThat(output).contains("Ergebnis: 4.000");
285 }
286
287 @Test
288 @DisplayName("Should show usage for too many arguments")
289 void testMainWithTooManyArguments() {
290     assertThrows(SystemExit.class, () -> {
291         try {
292             SquareRootCalculator.main(new String[]{"1", "2", "3", "4"});
293         } catch (Exception e) {
294             // Convert to our custom exception for testing
295             throw new SystemExit();
296         }
297     });
298
299     String output = outputStreamCaptor.toString();
300     assertThat(output).contains("Verwendung:");
301 }
302
303 @Test
304 @DisplayName("Should handle invalid number in main")
305 void testMainWithInvalidNumber() {
306     assertThrows(SystemExit.class, () -> {
307         try {
308             SquareRootCalculator.main(new String[]{"not_a_number"});
309         } catch (Exception e) {
310             throw new SystemExit();
311         }
312     });
313
314     String errorOutput = errorStreamCaptor.toString();
315     assertThat(errorOutput).contains("Fehler:");
316 }
317
318 @Test
319 @DisplayName("Should handle invalid precision in main")
320 void testMainWithInvalidPrecision() {
321     assertThrows(SystemExit.class, () -> {
322         try {
323             SquareRootCalculator.main(new String[]{"4", "not_a_number"});
324         } catch (Exception e) {
325             throw new SystemExit();
326         }
327     });
328
329     String errorOutput = errorStreamCaptor.toString();
330     assertThat(errorOutput).contains("Fehler:");
331 }
332
333 @Test
334 @DisplayName("Should handle interactive mode with empty precision")
335 void testInteractiveModeEmptyPrecision() throws Exception {
336     String input = "25\n\nquit\n";
337     System.setIn(new ByteArrayInputStream(input.getBytes()));
338 }

```

```

339     SquareRootCalculator.main(new String[]{});
340
341     String output = outputStreamCaptor.toString();
342     assertThat(output).contains("Eingabe: 25");
343     assertThat(output).contains("Nachkommastellen: 10");
344     assertThat(output).contains("Ergebnis: 5.0000000000");
345 }
346
347 @Test
348 @DisplayName("Should handle interactive mode with exit command")
349 void testInteractiveModeWithExit() throws Exception {
350     String input = "exit\n";
351     System.setIn(new ByteArrayInputStream(input.getBytes()));
352
353     SquareRootCalculator.main(new String[]{});
354
355     String output = outputStreamCaptor.toString();
356     assertThat(output).contains("Auf Wiedersehen!");
357 }
358
359 @Test
360 @DisplayName("Should handle interactive mode with invalid input")
361 void testInteractiveModeInvalidInput() throws Exception {
362     String input = "invalid\n5\nquit\n";
363     System.setIn(new ByteArrayInputStream(input.getBytes()));
364
365     SquareRootCalculator.main(new String[]{});
366
367     String output = outputStreamCaptor.toString();
368     String errorOutput = errorStreamCaptor.toString();
369
370     assertThat(errorOutput).contains("Fehler:");
371     assertThat(output).contains("Auf Wiedersehen!");
372 }
373
374 @Test
375 @DisplayName("Should show verification for small precision")
376 void testVerificationOutput() throws Exception {
377     SquareRootCalculator.main(new String[]{"4", "5"});
378
379     String output = outputStreamCaptor.toString();
380     assertThat(output).contains("Verifikation (Quadrat):");
381 }
382
383 @Test
384 @DisplayName("Should not show verification for high precision")
385 void testNoVerificationForHighPrecision() throws Exception {
386     SquareRootCalculator.main(new String[]{"4", "25"});
387
388     String output = outputStreamCaptor.toString();
389     assertThat(output).doesNotContain("Verifikation (Quadrat):");
390 }
391
392 @ParameterizedTest
393 @ValueSource(doubles = {0.05, 0.5, 5.0, 50.0})
394 @DisplayName("Should handle different number ranges for initial guess")
395 void testDifferentInitialGuessRanges(double input) throws Exception {
396     BigDecimal result = SquareRootCalculator.calculateSquareRoot(
397         BigDecimal.valueOf(input), 10);
398
399     assertThat(result).isNotNull();
400
401     // Verify accuracy by squaring
402     BigDecimal squared = result.multiply(result);
403     BigDecimal tolerance = new BigDecimal("0.0000000001");
404
405     assertThat(squared.subtract(BigDecimal.valueOf(input)).abs())
406         .isLessThanOrEqualTo(tolerance);
407 }

```

```

408
409 @Test
410 @DisplayName("Should test exception constructors")
411 void testExceptionConstructors() {
412     SquareRootCalculator.InvalidInputException invalidException =
413         new SquareRootCalculator.InvalidInputException(
414             "Test message");
415     assertThat(invalidException.getMessage()).isEqualTo("Test message");
416
417     SquareRootCalculator.CalculationException calcException =
418         new SquareRootCalculator.CalculationException(
419             "Calc error");
420     assertThat(calcException.getMessage()).isEqualTo("Calc error");
421 }
422
423 @Test
424 @DisplayName("Should handle convergence edge cases")
425 void testConvergenceEdgeCases() throws Exception {
426     // Test with a number that requires more iterations
427     BigDecimal result = SquareRootCalculator.calculateSquareRoot(
428         new BigDecimal("123456789012345678901234567890"), 20);
429
430     assertThat(result).isNotNull();
431     assertThat(result.scale()).isEqualTo(20);
432 }
433
434 @Test
435 @DisplayName("Should handle precision boundary values")
436 void testPrecisionBoundaryValues() throws Exception {
437     // Test with 0 decimal places
438     BigDecimal result = SquareRootCalculator.calculateSquareRoot(
439         new BigDecimal("4"), 0);
440     assertThat(result.scale()).isEqualTo(0);
441     assertThat(result.toString()).isEqualTo("2");
442
443     // Test with maximum precision
444     result = SquareRootCalculator.calculateSquareRoot(
445         new BigDecimal("4"), 1000);
446     assertThat(result.scale()).isEqualTo(1000);
447 }
448
449 // Helper class to simulate System.exit calls in tests
450 private static class SystemExit extends RuntimeException {
451 }
452 }

```

Beispielhafte Verwendung von Claude Sonnet 4 Thinking

Zusammenfassung

- Aufwand: ca. 30 Minuten (ca. 15 Minuten für den Prompt und ca. 15 Minuten für Claude Sonnet).
- Die Generierung musste explizit terminiert werden, da der Agent beim Ausführen der Anwendung nicht in der Lage war diese korrekt zu bedienen und in einer Endlos-Schleife hing.
- Der Code verwendet eigene Exception für Standardfälle. Dies ist ungeschickt.
- Projektstruktur und Tests entsprechen den Erwartungen.
- Die verwendeten Libraries waren auf den Ständen, die bei GPT nach der Aktualisierung verwendet wurden. D. h. nicht aktuell.
- Es wurden noch einige weitere Dateien generiert, die nicht gefordert wurden: README.md zum Beispiel mit Lizenzinformationen...
- Anwendung funktioniert und (viele) Tests auch - wenn man diese manuell ausführt.

Achtung aufgrund des Testens der main Methode welche `System.exit` Aufrufe hat kommt es zu Fehlern!

Durch einen weiteren Prompt mit dem Fehler konnte dies behoben werden. Dabei wurde auch gleich ein Test fachlich korrigiert (Thinking Model). Leider wurde auch das Build-Skript abgeändert und (wieder) veraltete Versionen der Tools eingetragen.

- Die Codeabdeckung ist gut, aber die meisten Sonderfälle werden nicht abgedeckt.
- Das Build-Skript funktionierte erst nicht, da die Versionen nicht korrekt ermittelt wurden!
- Der Code ist besser modularisiert in mehrere Methoden wodurch das Verständnis gefördert wird.
- Es wurden Hilfsmethoden für anzunehmende gängige Fälle implementiert.