SEQIS 10 things

SEQIS

... weil Softwaretest wichtig ist
100% Qualität
www.SEQIS.com

SEQIS 10 things

Herzlich Willkommen!

Alexander Weichselberger
SEQIS Geschäftsleitung

SEQIS 10 things – Programm 2014

- 20.03.14 Business Analyse
  Einführung in den BABOK® Guide
- 26.06.14 API Testing: Nur ein Schnittstelle
  und der passende Test
- **18.09.14 Test Driven Development – die Methode für Qualitätsbewusste**
- 20.11.14 Der Test als zentrale Schnittstelle
  in einem IT Projekt

Test Driven Development

Markus Zimmer, M.Sc.
Test Consultant

SEQIS

... weil Softwaretest wichtig ist
100% Qualität
www.SEQIS.com

Disclaimer

- TDD ist komplex
- Soll einen Überblick über die Kernthemen liefern
- Praktisches Beispiel ist zur Nachvollziehbarkeit einfach gehalten

Guter Code

- Erfüllt Requirements
- Fehlerfrei
- Wartbar
- Richtig dokumentiert
- Hohe Testabdeckung
- Hohe Qualität
- …

„Klassische“ Probleme in der Entwicklung

- Tests werden vernachlässigt oder ungenau gemacht
- Wartbarkeit des Codes nicht immer gewährleistet
- Änderungen an der Logik führen zu Fehler
- Lange Debugging Sessions wegen eines Fehlers
- …

Was ist TDD?

- Genau definierte Vorgehensweise
- Feste Phasen mit vorgegebenen Zielen
- Fokus auf die Anforderung

Ziele von TDD

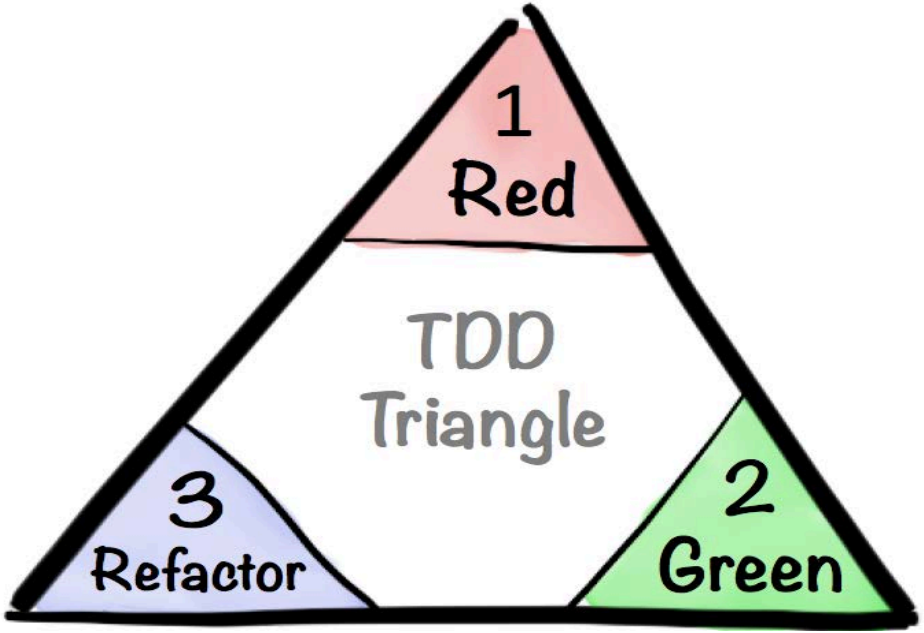
- Hohe Qualität des Codes
- Hohe Testabdeckung
- Wartbarkeit
- Vermeidung von Waste

1. Die Entscheidung für TDD ist noch keine Garantie für gute Qualität

- Entwickler muss sich auf TDD einstellen

Die Vorgehensweise von TDD

- Quelle: http://blog.crisp.se/wp-content/uploads/2013/10/Screen-Shot-2013-10-11-at-17.44.48.png

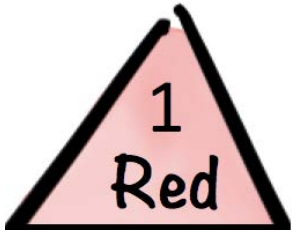


2. Halten Sie die einzelnen Phasen ein

- Jede Phase hat ein bestimmtes Ziel
- Keine Vermischung von Phasen!

Vorurteile von TDD

- Andere Denkweise notwendig
- Beginn von TDD ist holprig
- TDD ist viel aufwändiger



Tests zuerst!

- Keine Beeinflussung durch bestehenden Code
- Triangulation der Logik anstatt Overhead
- Jeder Test soll genau einen Zustand verifizieren
- FIRST

FIRST

- Ein Unittest muss folgende Kriterien erfüllen:
  - **F**ast
  - **I**solated and Intependent
  - **R**epeatable
  - **S**elf-Verifying
  - **T**imely

3. Wenden Sie FIRST an

- Kriterien, die ein Unittest erfüllen muss

Die Entwicklung der Logik

- Immer nur soviel Logik wie benötigt wird entwickeln
- Nicht vorarbeiten → Gefahr von Waste!
- Keine Logik bearbeiten die nichts mit dem aktuellen Test zu tun hat → Refactoring!

Refactoring

- Code wird überarbeitet
  - Umbenennung von Variablen, Methoden, Klassen, …
  - Extrahieren von Methoden, Klassen, …
- Keine neue Business Logik
- Muss nach jedem Schritt getestet werden

„Naming“ und „Duplication“

- Code muss korrekt benannt werden
  - Aussprechbarkeit
  - Selbsterklärend
- Duplizierter Code muss vermieden werden
  - „Single Point of Failure“

4. Naming und Duplication

- Kerndisziplinen der Software Entwicklung
- Einfach anzuwenden

5. Hinterfragen Sie Konventionen

- … und passen Sie sie an
- Kommunizieren Sie das Ergebnis

SOLID

- 5 Grundsätze für objektorientierte Programmierung
  - Single Responsibility
  - Open/Closed
  - Liskov Substitution
  - Interface Segregation
  - Dependency Inversion

Design Patterns

- Beste Vorgehensweise für Problemstellungen
- Sollen durch Refactoring implementiert werden
- Verschiedene Pattern mit unterschiedlichen Anwendungsgebieten
  - Strategy Pattern
  - Factory Pattern
  - Adapter Pattern
  - …

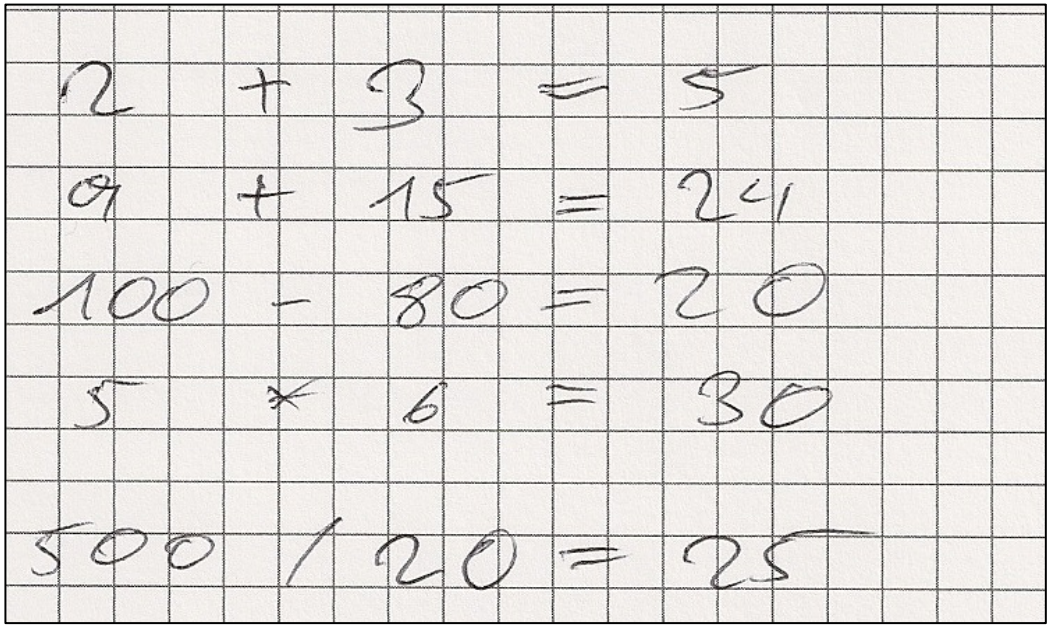
6. Verwenden Sie Design Patterns mit Bedacht

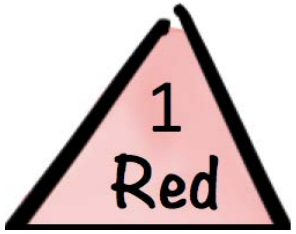
- Informieren Sie sich über Design Patterns
- Achtung: Vorsichtiger Einsatz

Praktisches Beispiel

- Implementierung eines Taschenrechner
- Einfache Grundrechnungsarten
- Schritt für Schritt Erklärung

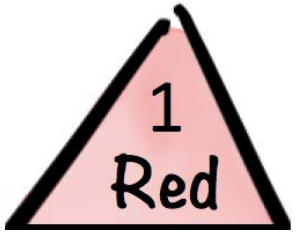
TODO Liste





Simple Addition als erster Test

```
10    @Test
11    public void Simple_Addition() {
12        Calculator calculator = new Calculator();
13        int returnValue = calculator.calculate("2+3");
14        assertEquals(5, returnValue);
15    }
```



Testergebnis

Runs: 1/1 Errors: 1 Failures: 0

com.seqis.tdd.CalculatorTests [Runner: JUnit 4] (0,005 s)

Simple_Addition (0,006 s)

Failure Trace

java.lang.Error: Unresolved compilation problems:

Calculator cannot be resolved to a type

Implementierung der Calculator Klasse

```
5⊖    public int calculate(String string) {
6         // TODO Auto-generated method stub
7         return 0;
8     }
```

Testergebnis

Finished after 0,015 seconds

Runs: 1/1 Errors: 0 Failures: 1

com.seqis.tdd.CalculatorTests [Runner: JUnit 4] (0,002 s)

Simple_Addition (0,002 s)

Failure Trace

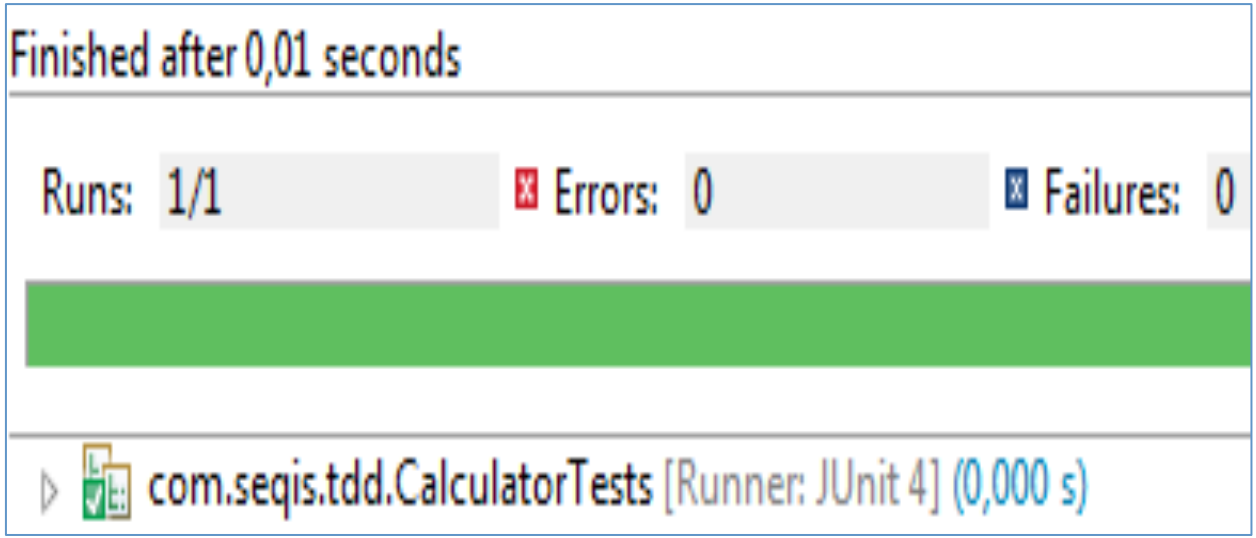
java.lang.AssertionError: expected:<5> but was:<0>

at com.seqis.tdd.CalculatorTests.Simple_Addition(CalculatorTests.java:14)

Implementierung der Logik

```
5⊖    public int calculate(String string) {
6         // TODO Auto-generated method stub
7         return 5;
8     }
```

Testergebnis



Calculator vor Refactoring

```
5⊖    public int calculate(String string) {
6         // TODO Auto-generated method stub
7         return 0;
8     }
```

Refactoring von Calculation

```
5⊖    public int calculate(String calculation) {
6         return 5;
7     }
```

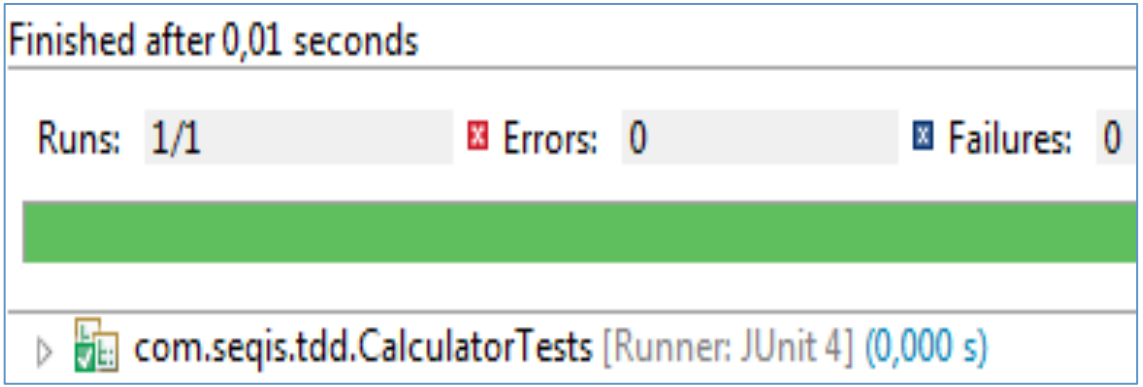
Testklasse vor Refactoring

```
10    @Test
11    public void Simple_Addition() {
12        Calculator calculator = new Calculator();
13        int returnValue = calculator.calculate("2+3");
14        assertEquals(5, returnValue);
15    }
```

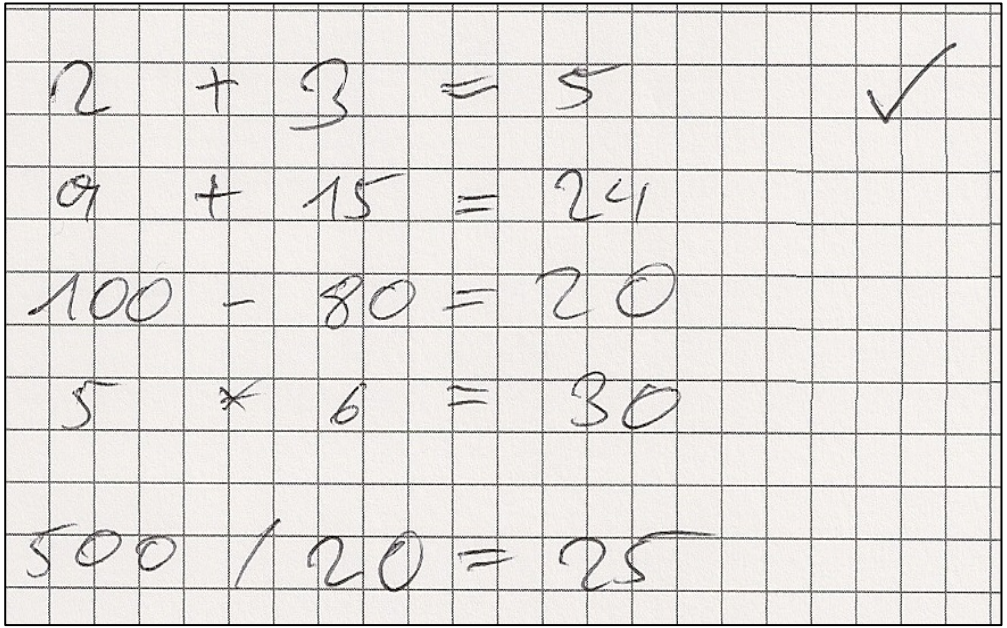
Refactoring von Testklasse

```
10⊖    @Test
11     public void Simple_Addition() {
12         Calculator calculator = new Calculator();
13         int result = calculator.calculate("2+3");
14         assertEquals(5, result);
15     }
```

Überprüfung ob Refactoring korrekt war



TODO

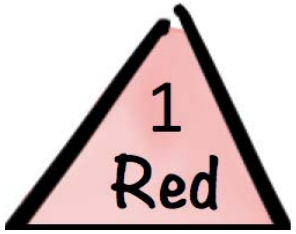


7. Machen Sie kleine Schritte

- Nur umsetzen was aktuell(!) gebraucht wird
- Bei Refactoring wird nach jedem Schritt getestet

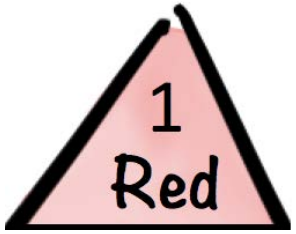
8. Refactoring Methoden der Entwicklungsumgebung nutzen

- Entwicklungsumgebungen vereinfachen den gesamten Vorgang erheblich



Weitere Addition

```
17    @Test
18    public void Another_Simple_Addition() {
19        Calculator calculator = new Calculator();
20        int result = calculator.calculate("9+15");
21        assertEquals(24, result);
22    }
```



Testergebnis

Runs: 2/2 Errors: 0 Failures: 1

com.seqis.tdd.CalculatorTests [Runner: JUnit 4] (0,002 s)
Simple_Addition (0,000 s)
Another_Simple_Addition (0,002 s)

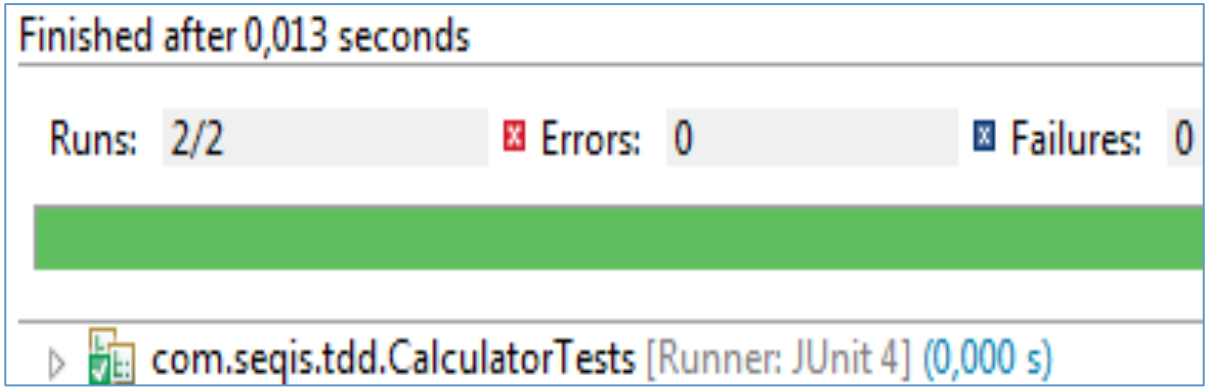
Failure Trace

java.lang.AssertionError: expected:<24> but was:<5>

Erweiterung der Additionslogik

```
5   public int calculate(String calculation) {
6       String[] operands = calculation.split("\\+");
7       return Integer.valueOf(operands[0]) + Integer.valueOf(operands[1]);
8   }
```

Testergebnis



Logik vor Überarbeitung

```
5    public int calculate(String calculation) {
6        String[] operands = calculation.split("\\+");
7        return Integer.valueOf(operands[0]) + Integer.valueOf(operands[1]);
8    }
```

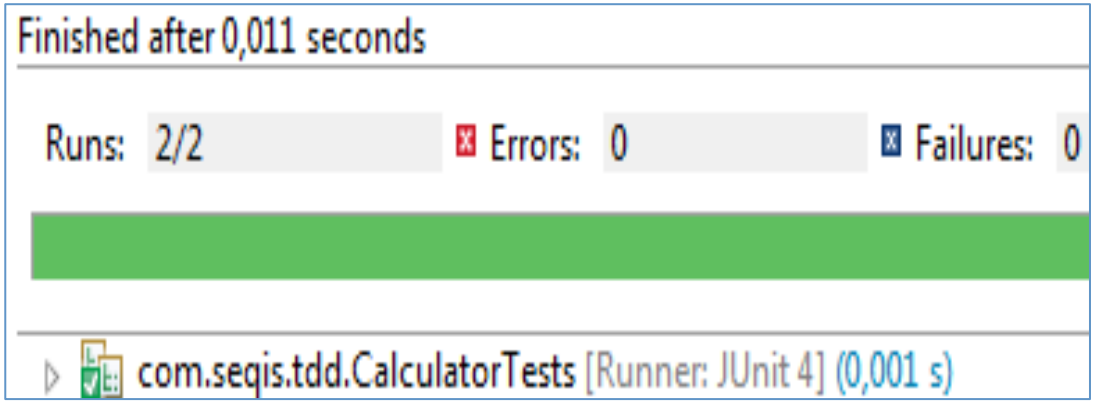
Auslagerung von Logik

```
5⊖    public int calculate(String calculation) {
6         String[] operands = calculation.split("\\+");
7         return convertToNumber(operands[0]) + convertToNumber(operands[1]);
8     }
9
10⊖   private int convertToNumber(String operand) {
11        return Integer.valueOf(operand);
12    }
```

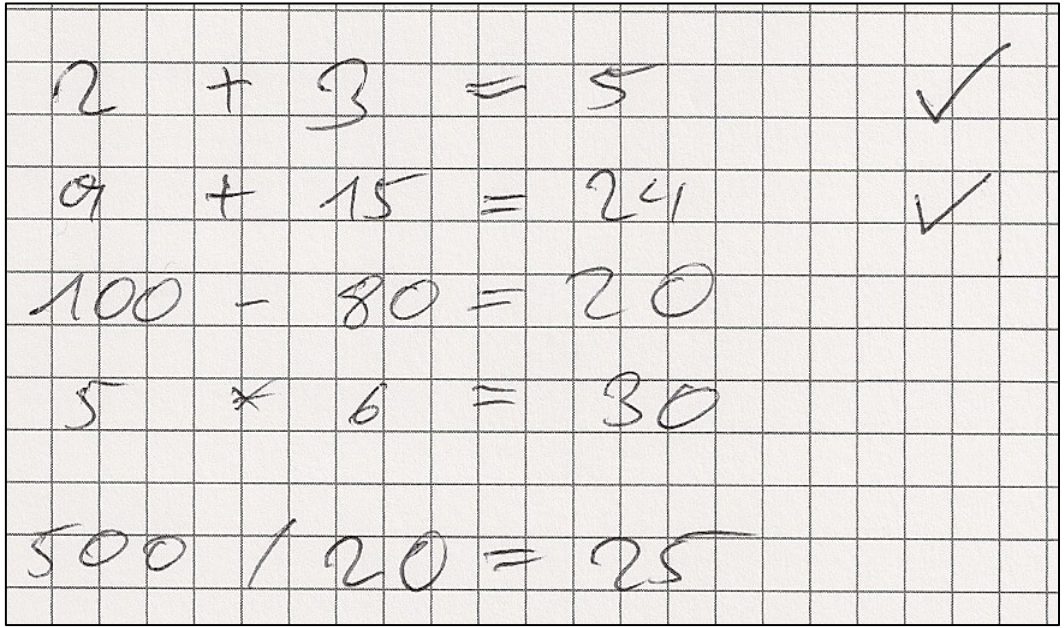
Überarbeitung der Testfälle

```
10    public void calculateAndAssert(String calculation, int expectedResult) {
11        Calculator calculator = new Calculator();
12        int result = calculator.calculate(calculation);
13        assertEquals(expectedResult, result);
14    }
15
16    @Test
17    public void Simple_Addition() {
18        calculateAndAssert("2+3", 5);
19    }
20
21    @Test
22    public void Another_Simple_Addition() {
23        calculateAndAssert("9+15", 24);
24    }
```

Testergebnis

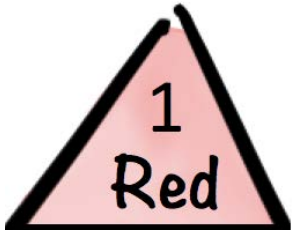


TODO



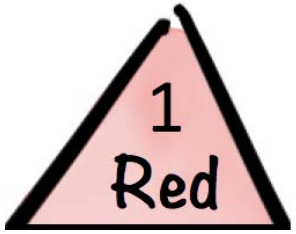
9. Pflegen Sie Ihre Testfälle

- Living Documentation“
- Gute Testfälle sind selbsterklärend



Simple Subtraktion

```
@Test
public void Simple_Subtraction() {
    calculateAndAssert("100-80", 20);
}
```



Testergebnis

Runs: 3/3 Errors: 1 Failures: 0

com.seqis.tdd.CalculatorTests [Runner: JUnit 4] (0,002 s)
Simple_Addition (0,000 s)
Another_Simple_Addition (0,000 s)
Simple_Substraction (0,002 s)

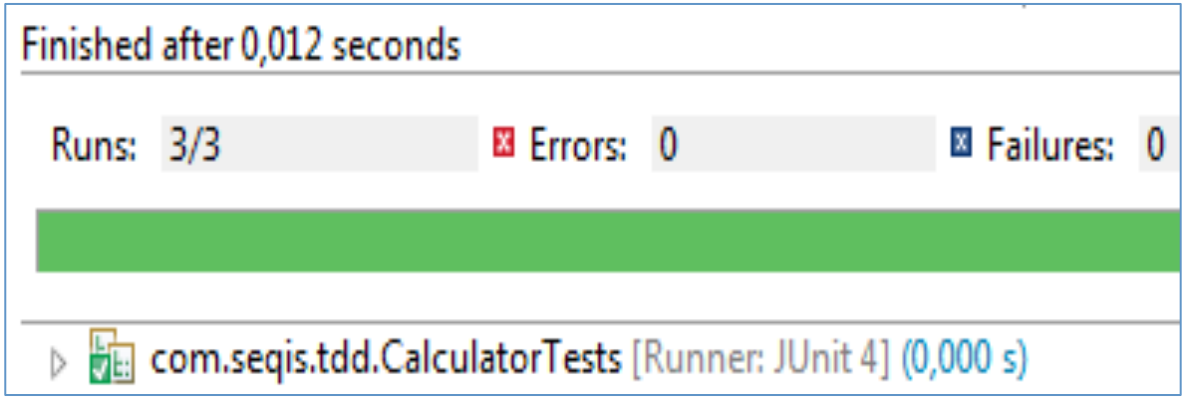
Failure Trace

java.lang.NumberFormatException: For input string: "100-80"

Implementierung der notwendigen Logik

```
5⊖    public int calculate(String calculation) {
6         if(calculation.contains("+")) {
7             String[] operands = calculation.split("\\+");
8             return convertToNumber(operands[0]) + convertToNumber(operands[1]);
9         } else {
10            String[] operands = calculation.split("\\-");
11            return convertToNumber(operands[0]) - convertToNumber(operands[1]);
12        }
13    }
```

Testergebnis



Refactoring – Strategy Pattern

```
3  public abstract class CalculationStrategy {
4
5      protected String REGEX_MASKING = "\\";
6
7      protected abstract String getOperator();
8      protected abstract int calculate(int firstNumber, int secondNumber);
9
10     protected int convertToNumber(String operand) {
11         return Integer.valueOf(operand);
12     }
13
14     protected String[] getOperands(String calculation) {
15         return calculation.split(REGEX_MASKING + getOperator());
16     }
17
18     public boolean canCalculate(String calculation) {
19         return calculation.contains(getOperator());
20     }
21
22     public int processCalculation(String calculation) {
23         String[] operands = getOperands(calculation);
24         return calculate(convertToNumber(operands[0]), convertToNumber(operands[1]));
25     }
26
27 }
```

Implementierung der Additionslogik

```
3  public class Addition extends CalculationStrategy {
4
5      @Override
6      protected String getOperator() {
7          return "+";
8      }
9
10     @Override
11     protected int calculate(int firstNumber, int secondNumber) {
12         return firstNumber + secondNumber;
13     }
14
15 }
```

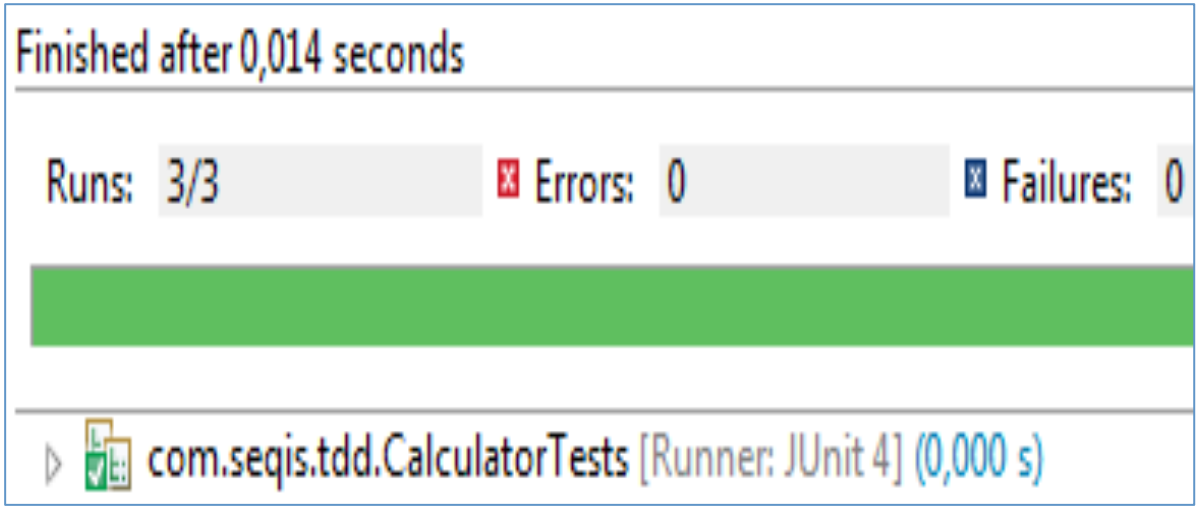
Verwendung

```
 6 public class Calculator {
 7
 8     List<CalculationStrategy> calculationStrategies;
 9
10     public static Calculator getInstance() {
11         return new Calculator(new Addition(), new Subtraction());
12     }
13
14     protected Calculator (CalculationStrategy... strategies) {
15         calculationStrategies = Arrays.asList(strategies);
16     }
17
18     public int calculate(String calculation) throws Exception {
19         for(CalculationStrategy calculationStrategy : calculationStrategies) {
20             if(calculationStrategy.canCalculate(calculation)) {
21                 return calculationStrategy.processCalculation(calculation);
22             }
23         }
24         throw new Exception("Unsupported calculation!");
25     }
26
27 }
```

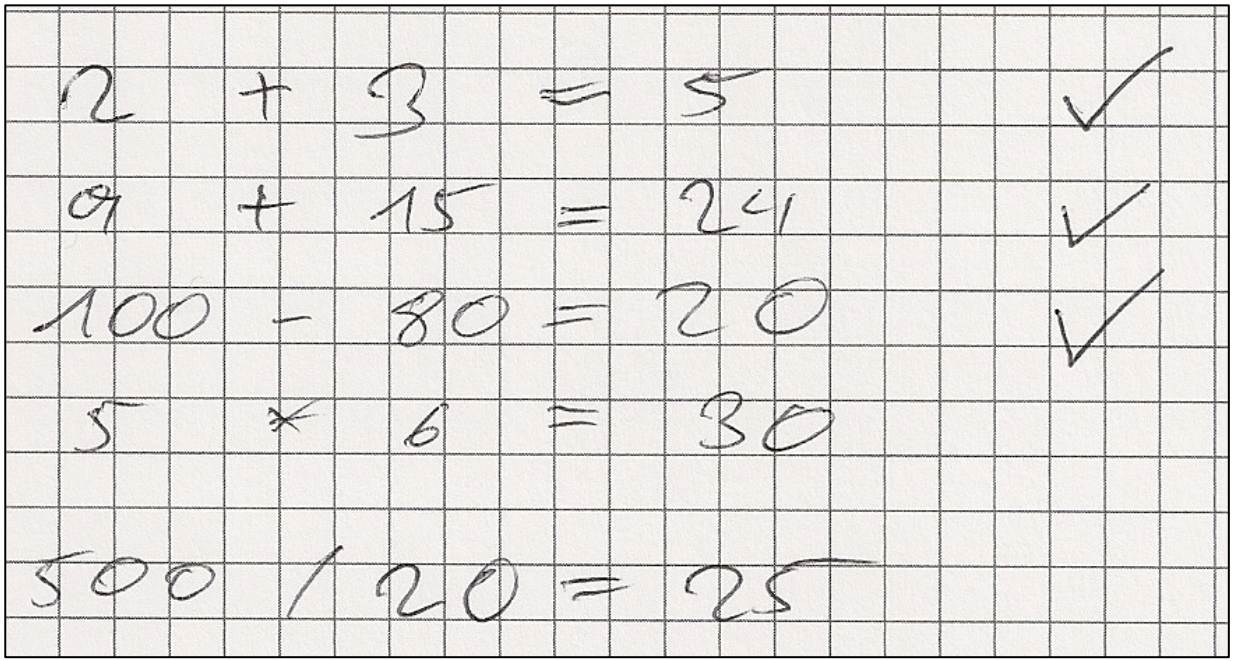
Testfallüberarbeitung

```
11    public void calculateAndAssert(String calculation, int expectedResult) {
12        Calculator calculator = Calculator.getInstance();
13        int result;
14        try {
15            result = calculator.calculate(calculation);
16            assertEquals(expectedResult, result);
17        } catch (Exception e) {
18            Assert.fail(e.getMessage());
19        }
20    }
```

Testergebnis



TODO



10. Refactoring kann umfangreich sein

- Viel Veränderung ist ok

Vorteile von TDD

- Überblick durch TODO Liste gegeben
- Einzelne Iterationen lassen sich schnell umsetzen
- Sicherheitsnetz durch Tests
- Richtig angewandt entsteht kein Waste

Zusammenfassung

1. Die Entscheidung für TDD ist noch keine Garantie für gute Qualität
2. Halten Sie die einzelnen Phasen ein
3. Wenden Sie FIRST an
4. Naming und Duplication
5. Hinterfragen Sie Konventionen
6. Verwenden Sie Design Patterns mit Bedacht
7. Machen Sie kleine Schritte
8. Refactoring Methoden der Entwicklungsumgebung nutzen
9. Pflegen Sie Ihre Testfälle
10. Refactoring kann Umfangreich sein

SEQIS 10 things
Test Driven Development

Markus Zimmer, M.Sc.
Test Consultant

SEQIS

... weil Softwaretest wichtig ist ★ www.SEQIS.com ★ 100% Qualität

SEQIS 10 things – Programm 2014

- 20.03.14 Business Analyse
  Einführung in den BABOK® Guide
- 26.06.14 API Testing: Nur ein Schnittstelle
  und der passende Test
- 18.09.14 Test Driven Development – die
  Methode für Qualitätsbewusste
- **20.11.14 Der Test als zentrale Schnittstelle**
  in einem IT Projekt