Test Driven Development mit Java - Beispiele

Teil 2

Gesamtinhaltsverzeichnis

1

Einleitung

1 Einleitung

In diesem Skript werden drei nicht triviale Aufgaben beschrieben, deren Lösung sich an der Idee der testgetriebenen Software-Entwicklung orientiert.

- Im ersten Beispiel geht's um die Entwicklung eines Scanners und eines Parsers für numerische Expressions.
- Im zweiten Beispiel geht's um die Entwicklung eines Systems zum Entwurf logischer Schaltungen.
- Beim dritten Beispiel handelt es sich um das Einlesen von CSV-Dateien. Die Zeilen solcher Dateien werden transformiert in Java-Objekte.

Das erste Beispiel ist fokussiert auf das Konzept einzelner, kleiner Schritte. Im zweiten Beispiel steht eher der Spezifikations-Aspekt im Vordergrund. Im dritten Beispiel schließlich geht's u.a. um die Verwendung von Reflection.

2

Parser

2 Parser

Die Vorgaben für diese Übung und ein erster Einstieg befinden sich im Hauptscript. Sie sind im Kapitel 8, Abschnitt 1 beschrieben.

2.1 Scanner – Whitespace

Der `Scanner` soll Operator-Zeichen natürlich auch dann erkennen, wenn diese in Whitespace-Zeichen eingebettet sind. Alle bisherigen Methoden des `ScannerTests` sollten bestehen bleiben. Die Testklasse wird durch eine neue Methode `testWhitespace` erweitert, in welcher dem `Scanner` ein "komplizierterer" String übergeben wird: `" + -* / "`

Die Erweiterung der Klasse `ScannerTest`:

```
package test;
// ...
public class ScannerTest {

    // ...

    @Test
    public void testWhitespace() {
        final Scanner scanner =
            new ScannerImpl(new StringReader(" + -  */  "));
        scanner.next();
        Assert.assertSame(Symbol.PLUS, scanner.currentSymbol());
        scanner.next();
        Assert.assertSame(Symbol.MINUS, scanner.currentSymbol());
        scanner.next();
        Assert.assertSame(Symbol.TIMES, scanner.currentSymbol());
        scanner.next();
        Assert.assertSame(Symbol.DIV, scanner.currentSymbol());
        scanner.next();
        Assert.assertNull(scanner.currentSymbol());
    }
```

Die Erweiterung der `ScannerImpl`-Klasse ist in diesem Schritt recht einfach:

```
package scanner;

public class ScannerImpl implements Scanner {

    // ...

    @Override
    public void next() {
        while(Character.isWhitespace(this.currentChar))
            this.nextChar();
        if (this.currentChar == -1) {
            this.currentSymbol = null;
            return;
        }
        this.currentSymbol = operators.get((char)this.currentChar);
        if (this.currentSymbol == null)
            throw new RuntimeException("unexpected char: " +
                (char)this.currentChar +
                 " code = " + this.currentChar);
        this.nextChar();
    }

    // ...
}
```

2.2 Scanner – Numbers

Der Scanner soll Zahlen erkennen können: sowohl Ganzzahlen als auch Gleitkomma-Zahlen.

Ihm wird z.B. folgender String übergeben: `" + 1 3.14 - * 300/ "`. (Natürlich ist die Reihenfolge dieser Symbolfolge "unsinnig" – aber über Sinn resp. Unsinn urteilt der Scanner nicht…)

Die Erweiterung des `ScannerTests`:

```
package test;
// ...
public class ScannerTest {

    // ...

    @Test
    public void testNumbers() {
        final Scanner scanner = new ScannerImpl(
            new StringReader(" +  1 3.14 - *  300/ "));
        scanner.next();
        Assert.assertEquals(
            Symbol.PLUS, scanner.currentSymbol());
        scanner.next();
        Assert.assertEquals(
            Symbol.NUMBER, scanner.currentSymbol());
        Assert.assertEquals(1.0, scanner.getNumber(), 0);
        scanner.next();
        Assert.assertEquals(
            Symbol.NUMBER, scanner.currentSymbol());
        Assert.assertEquals(3.14, scanner.getNumber(), 0);
        scanner.next();
        Assert.assertEquals(
            Symbol.MINUS, scanner.currentSymbol());
        scanner.next();
        Assert.assertEquals(
            Symbol.TIMES, scanner.currentSymbol());
        scanner.next();
        Assert.assertEquals(
            Symbol.NUMBER, scanner.currentSymbol());
        Assert.assertEquals(300, scanner.getNumber(), 0);
        scanner.next();
        Assert.assertEquals(
            Symbol.DIV, scanner.currentSymbol());
        scanner.next();
        Assert.assertNull(scanner.currentSymbol());
    }
```

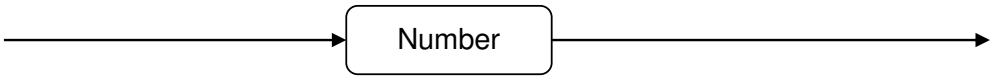
Die Erweiterung der `ScannerImpl`-Klasse:

```
package scanner;

public class ScannerImpl implements Scanner {

    private double currentNumber;

    // ...

    @Override
    public void next() {
        while (Character.isWhitespace(this.currentChar))
            this.nextChar();
        if (this.currentChar == -1) {
            this.currentSymbol = null;
            return;
        }
        if (Character.isDigit(this.currentChar)) {
            final StringBuilder buf = new StringBuilder();
            buf.append((char) this.currentChar);
            this.nextChar();
            while (Character.isDigit(this.currentChar)) {
                buf.append((char) this.currentChar);
                this.nextChar();
            }
            if (this.currentChar == '.') {
                buf.append('.');
                this.nextChar();
                while (Character.isDigit(this.currentChar)) {
                    buf.append((char) this.currentChar);
                    this.nextChar();
                }
            }
            this.currentNumber = Double.parseDouble(buf.toString());
            this.currentSymbol = Symbol.NUMBER;
            return;
        }
        this.currentSymbol = operators.get((char) this.currentChar);
        if (this.currentSymbol == null)
            throw new RuntimeException("unexpected char: " +
                (char) this.currentChar +
                " code = " + this.currentChar);
        this.nextChar();
    }

    @Override
    public double getNumber() {
        return this.currentNumber;
    }

    // ...
}
```

2.3 Parsing Numbers

Der einfachste Ausdruck, dessen Wert berechnet werden kann, ist eine Zahl. Beginnen wird daher mit einem sehr einfachen Parser, der unter Zuhilfenahme des Scanners eine Zahl einliest und deren Wert zurückliefert.

Expression:



Zunächst auch hier wieder die Spezifikation des Parsers:

```
package parser;

public interface DoubleParser {
    public abstract double parse();
}
```

Dann die Testklasse (die der Compiler aber noch nicht vollständig übersetzt, weil die Parser-Implementierung im Augenblick noch fehlt):

```
package test;

import java.io.StringReader;

import org.junit.Assert;
import org.junit.Test;

import parser.DoubleParser;
import parser.DoubleParserImpl;
import scanner.ScannerImpl;
import util.test.XAssert;

public class DoubleParserTest {

    @Test
    public void test() {
        final DoubleParser parser = new DoubleParserImpl(
            new ScannerImpl(new StringReader(" 2.71  ")));
        Assert.assertEquals(2.71, parser.parse(), 0);
    }

    @Test
    public void testEmpty() {
        final DoubleParser parser = new DoubleParserImpl(
            new ScannerImpl(new StringReader(" ")));
        XAssert.assertThrows(RuntimeException.class,
            () -> parser.parse());
    }

    @Test
    public void testTooManyNumbers() {
        final DoubleParser parser = new DoubleParserImpl(
            new ScannerImpl(new StringReader(" 2.71 3.14")));
        XAssert.assertThrows(RuntimeException.class,
            () -> parser.parse());
    }
}
```

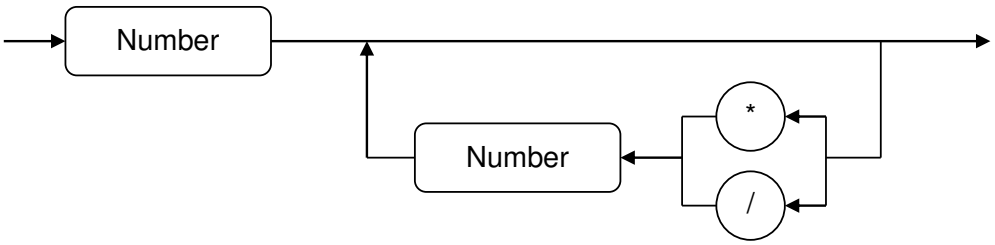
Die Implementierung des Parsers:

```
package parser;

import scanner.Scanner;
import scanner.Symbol;

public class DoubleParserImpl implements DoubleParser {

    private final Scanner scanner;

    public DoubleParserImpl(final Scanner scanner) {
        this.scanner = scanner;
        this.scanner.next();
    }

    @Override
    public double parse() {
        if (this.scanner.currentSymbol() != Symbol.NUMBER)
            throw new RuntimeException("Number expected");
        final double value = this.scanner.getNumber();
        this.scanner.next();
        if (this.scanner.currentSymbol() != null)
            throw new RuntimeException("EOF expected");
        return value;
    }
}
```

2.4 Parsing multiplicative Expressions

Ein Ausdruck kann ein multiplikativer Ausdruck sein: eine Zahl gefolgt von einem multiplikativen Operator (*, /) gefolgt von einer Zahl gefolgt von….

Multiplicative:



Die Erweiterung des Tests:

```
package test;
// ...
public class DoubleParserTest {

    // ...

    @Test
    public void testMultiplicative() {
        final DoubleParser parser = new DoubleParserImpl(
            new ScannerImpl(new StringReader(" 2 * 3 / 4 ")));
        Assert.assertEquals(1.5, parser.parse(), 0);
    }
}
```

Die Erweiterung der Parser-Implementierung:

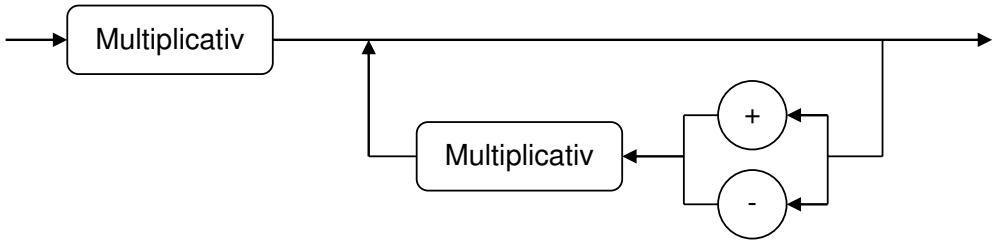
```
package parser;
// ...
public class DoubleParserImpl implements DoubleParser {

    // ...

    @Override
    public double parse() {
        final double value = this.parseMultiplicative();
        if (this.scanner.currentSymbol() != null)
            throw new RuntimeException("EOF expected");
        return value;
    }

    public double parseMultiplicative() {
        double value = this.parseNumber();
        while (this.scanner.currentSymbol() == Symbol.TIMES
                || this.scanner.currentSymbol() == Symbol.DIV) {
            final boolean times =
                this.scanner.currentSymbol() == Symbol.TIMES;
            this.scanner.next();
            final double v = this.parseNumber();
            if (times)
                value = value * v;
            else
                value = value / v;
        }
        return value;
    }

    public double parseNumber() {
        if (this.scanner.currentSymbol() != Symbol.NUMBER)
            throw new RuntimeException("Number expected");
        final double value = this.scanner.getNumber();
        this.scanner.next();
        return value;
    }
}
```

2.5 Parsing additive Expressions

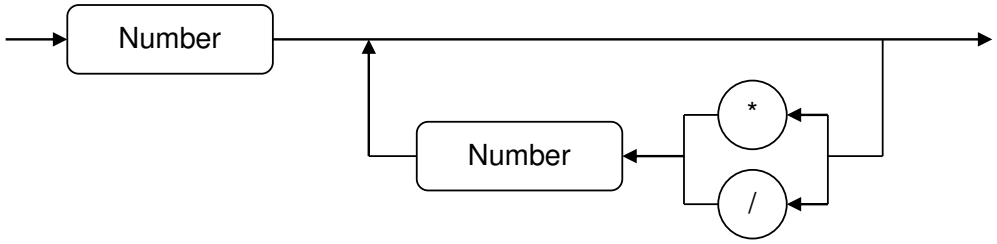
Ein Ausdruck kann ein multiplikativer Ausdruck sein: ein "Term" gefolgt von einem multiplikativen Operator (*, /) gefolgt von einem "Term" gefolgt von….

Und was ist ein "Term"? Ein Term ist ein additiver Ausdruck (s.o.)

Additive:



Multiplicative:



Die Erweiterung der Testklasse:

```
package test;
// ...
public class DoubleParserTest {

    // ...

    @Test
    public void testAdditive() {
        final DoubleParser parser = new DoubleParserImpl(
            new ScannerImpl(new StringReader(" 2 + 12 / 4 - 2 * 5")));
        Assert.assertEquals(-5.0, parser.parse(), 0);
    }

}
```

Die Erweiterung der Implementierung:

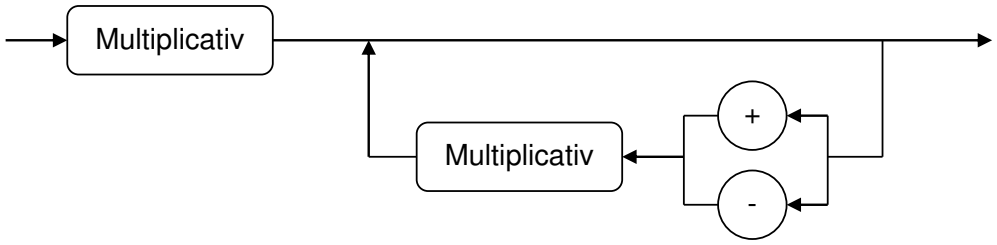
```
package parser;
// ...
public class DoubleParserImpl implements DoubleParser {
    // ...

    @Override
    public double parse() {
        final double value = this.parseAdditive();
        if (this.scanner.currentSymbol() != null)
            throw new RuntimeException("EOF expected");
        return value;
    }

    public double parseAdditive() {
        double value = this.parseMultiplicative();
        while (this.scanner.currentSymbol() == Symbol.PLUS
                || this.scanner.currentSymbol() == Symbol.MINUS) {
            final boolean plus =
                this.scanner.currentSymbol() == Symbol.PLUS;
            this.scanner.next();
            final double v = this.parseMultiplicative();
            if (plus)
                value = value + v;
            else
                value = value - v;
        }
        return value;
    }

    public double parseMultiplicative() {
        // ...
    }

    public double parseNumber() {
        // ...
    }
}
```

2.6 Parsing rekursive Expressions

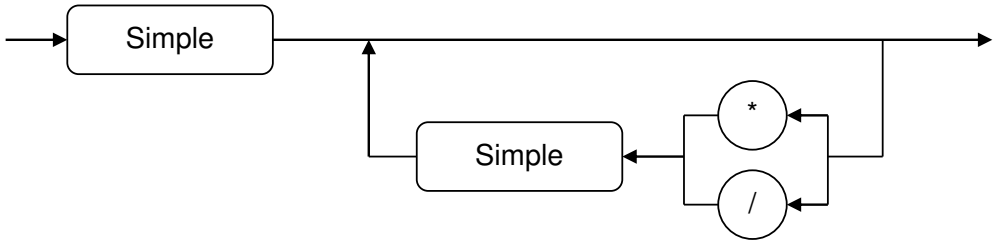
Ein multiplikativer Ausdruck war bislang definiert als eine multiplikative Verknüpfung von Zahlen. Dort aber, wo eine Zahl stehen kann, kann natürlich auch ein weiterer, geklammerter Ausdruck stehen.

Wir korrigieren (bzw. erweitern) also: ein multiplikativer Ausdruck ist eine multiplikative Verknüpfung von Simple-Expression. Und eine Simple Expression ist entweder eine Zahl oder aber ein geklammerter additiver Ausdruck. Die ganze Definition ist somit rekursiv geworden.

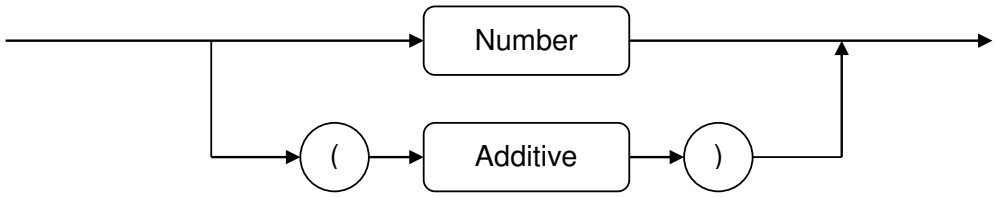
Additive:



Multiplicative:



Simple:



Die Erweiterung des Tests:

```
package test;
// ...
public class DoubleParserTest {

    // ...

    @Test
    public void testNested() {
        final DoubleParser parser = new DoubleParserImpl(
            new ScannerImpl(new StringReader(
                " (2 + 58) / (4 - (2-1)) * 5")));
        Assert.assertEquals(100.0, parser.parse(), 0);
    }
}
```

Die Erweiterung der Implementierung:

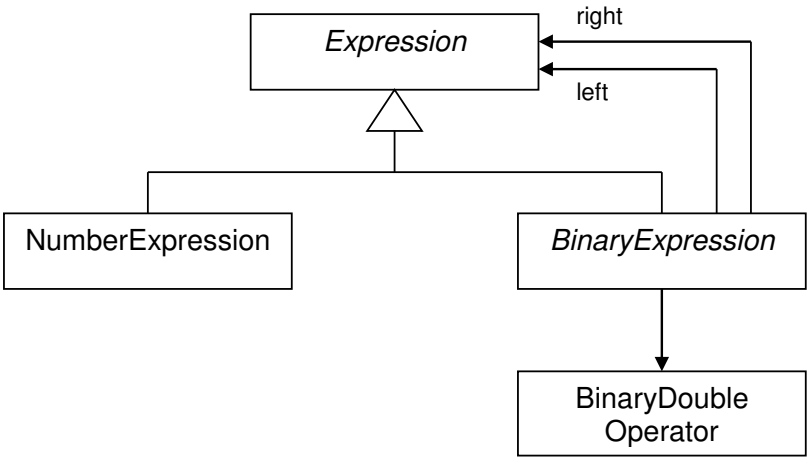
```
package parser;
// ...
public class DoubleParserImpl implements DoubleParser {

    // ...

    public double parseMultiplicative() {
        double value = this.parseSimple();
        while (this.scanner.currentSymbol() == Symbol.TIMES
                || this.scanner.currentSymbol() == Symbol.DIV) {
            final boolean times =
                this.scanner.currentSymbol() == Symbol.TIMES;
            this.scanner.next();
            final double v = this.parseSimple();
            if (times)
                value = value * v;
            else
                value = value / v;
        }
        return value;
    }

    public double parseSimple() {
        if(this.scanner.currentSymbol() == Symbol.NUMBER) {
            return this.parseNumber();
        }
        if (this.scanner.currentSymbol() == Symbol.OPEN) {
            this.scanner.next();
            final double value = this.parseAdditive();
            if (this.scanner.currentSymbol() != Symbol.CLOSE)
                throw new RuntimeException(") expected");
            this.scanner.next();
            return value;
        }
        throw new RuntimeException("Number or ( expected");
    }

    public double parseNumber() { ... }
    }
}
```

2.7 Expressions

Der `Parser` soll keinen Wert berechnen, sondern einen Baum von Expressions liefern. Zuvor wurde immer von Expression gesprochen, ohne aber dieses Konzept in Klassen zu implementieren.

Eine `Expression` ist entweder eine `NumberExpression` oder eine `BinaryExpression`. Eine `BinaryExpression` verweist auf ihre beiden Seiten (`left` und `right`), die wiederum eine `Expression` sind. Eine `BinaryExpression` besitzt zudem einen `DoubleBinaryOperator`.

`Expression` spezifiziert eine Methode `double evaluate()`, welche von den abgeleiteten Klassen natürlich implementiert werden muss.



Natürlich sollte eine neue Testklasse eingeführt werden, die sich ausschließlich mit dem Testen von Expressions befasst. Da diese Klassen aber recht einfach sind, verzichten wir ausnahmsweise auf den Test…

Das Interface `Expression`:

```
package expressions;

public abstract class Expression {
    public abstract double evaluate();
}
```

Die Klasse `NumberExpression`:

```
package expressions;

public class NumberExpression extends Expression {

    private final double value;

    public NumberExpression(final double value) {
        this.value = value;
    }

    @Override
    public double evaluate() {
        return this.value;
    }
}
```

Die Klasse `BinaryExpression`:

```
package expressions;

import java.util.function.DoubleBinaryOperator;

public class BinaryExpression extends Expression {

    private final Expression left;
    private final DoubleBinaryOperator op;
    private final Expression right;

    public BinaryExpression(
            final Expression left,
            final DoubleBinaryOperator op,
            final Expression right) {
        this.left = left;
        this.op = op;
        this.right = right;
    }

    @Override
    public double evaluate() {
        return this.op.applyAsDouble(
            this.left.evaluate(), this.right.evaluate());
    }
}
```

Für den Parser definieren wir nun ein neues Interface:

```
package parser;

import expressions.Expression;

public interface ExpressionParser {
    public abstract Expression parse();
}
```

Der Test der Implementierungs-Klasse (wir benutzten den `ValueParserTest` als Grundlage):

```
package test;
// ...
public class ExpressionParserTest {

    @Test
    public void test() {
        final ExpressionParser parser =
            new ExpressionParserImpl(
                new ScannerImpl(new StringReader(" 2.71  ")));
        Assert.assertEquals(
            2.71, parser.parse().evaluate(), 0);
    }

    @Test
    public void testEmpty() {
        final ExpressionParser parser =
            new ExpressionParserImpl(
                new ScannerImpl(new StringReader(" ")));
        XAssert.assertThrows(
            RuntimeException.class, () -> parser.parse());
    }

    @Test
    public void testTooManyNumbers() {
        final ExpressionParser parser =
            new ExpressionParserImpl(
                new ScannerImpl(new StringReader(" 2.71 3.14")));
        XAssert.assertThrows(
            RuntimeException.class, () -> parser.parse());
    }

    @Test
    public void testMultiplicative() {
        final ExpressionParser parser =
            new ExpressionParserImpl(
                new ScannerImpl(new StringReader(" 2 * 3 / 4 ")));
        Assert.assertEquals(
            1.5, parser.parse().evaluate(), 0);
    }

    @Test
    public void testAdditive() {
        final ExpressionParser parser =
            new ExpressionParserImpl(
                new ScannerImpl(
                    new StringReader(" 2 + 12 / 4 - 2 * 5")));
        Assert.assertEquals(
            -5.0, parser.parse().evaluate(), 0);
    }

    @Test
    public void testNested() {
        final ExpressionParser parser =
            new ExpressionParserImpl(
                new ScannerImpl(
                    new StringReader(" (2 + 58) / (4 - (2-1)) * 5")));
        Assert.assertEquals(
            100.0, parser.parse().evaluate(), 0);
    }
}
```

Die Implementierung des neuen Parsers übernehmen wir zum Großteil von der `ValueParserImpl`-Klasse. Wir ersetzen `double` durch `Expression` – und dort, wo bislang aufgrund jeweils eines binären Operators ein neuer `double` berechnet wurde, wird nun eine neue `Expression` erzeugt:

```
package parser;
// ...
public class ExpressionParserImpl implements ExpressionParser {

    private final Scanner scanner;

    public ExpressionParserImpl(final Scanner scanner) {
        this.scanner = scanner;
        this.scanner.next();
    }

    @Override
    public Expression parse() {
        final Expression value = this.parseAdditive();
        if (this.scanner.currentSymbol() != null)
            throw new RuntimeException("EOF expected");
        return value;
    }

    public Expression parseAdditive() {
        Expression value = this.parseMultiplicative();
        while (this.scanner.currentSymbol() == Symbol.PLUS
                || this.scanner.currentSymbol() == Symbol.MINUS) {
            final boolean plus =
                this.scanner.currentSymbol() == Symbol.PLUS;
            this.scanner.next();
            final Expression v = this.parseMultiplicative();
            if (plus)
                value = new BinaryExpression(
                    value, (x, y) -> x + y, v);
            else
                value = new BinaryExpression(
                    value, (x, y) -> x - y, v);
        }
        return value;
    }

    public Expression parseMultiplicative() {
        Expression value = this.parseSimple();
        while (this.scanner.currentSymbol() == Symbol.TIMES
                || this.scanner.currentSymbol() == Symbol.DIV) {
            final boolean times =
                this.scanner.currentSymbol() == Symbol.TIMES;
            this.scanner.next();
            final Expression v = this.parseSimple();
            if (times)
                value = new BinaryExpression(
                    value, (x, y) -> x * y, v);
            else
                value = new BinaryExpression(
                    value, (x, y) -> x / y, v);
        }
```

```
        return value;
    }

    public Expression parseSimple() {
        if(this.scanner.currentSymbol() == Symbol.NUMBER) {
            return this.parseNumber();
        }
        if (this.scanner.currentSymbol() == Symbol.OPEN) {
            this.scanner.next();
            final Expression value = this.parseAdditive();
            if (this.scanner.currentSymbol() != Symbol.CLOSE)
                throw new RuntimeException(") expected");
            this.scanner.next();
            return value;
        }
        throw new RuntimeException("Number or ( expected");
    }

    public Expression parseNumber() {
        if (this.scanner.currentSymbol() != Symbol.NUMBER)
            throw new RuntimeException("Number expected");
        final double value = this.scanner.getNumber();
        this.scanner.next();
        return new NumberExpression(value);
    }
}
```

2.8 Ein generischer Parser

Ein `DoubleParserImpl` liefert als Ergebnis von `parse` einen `double`-Wert; ein `ExpressionParserImpl` liefert eine `Expression`.

Der im Folgenden entwickelte generische Parser liefert ein `T`. Hier die Spezifikation:

```
package parser;

public interface Parser<T> {
    public abstract T parse();
}
```

Die Implementierung (sie ist abstrakt):

```
package parser;

import java.util.function.DoubleBinaryOperator;

import scanner.Scanner;
import scanner.Symbol;

public abstract class AbstractParser<T> implements Parser<T> {

    private final Scanner scanner;

    public AbstractParser(final Scanner scanner) {
        this.scanner = scanner;
        this.scanner.next();
    }

    @Override
    public T parse() {
        final T value = this.parseAdditive();
        if (this.scanner.currentSymbol() != null)
            throw new RuntimeException("EOF expected");
        return value;
    }

    public T parseAdditive() {
        T value = this.parseMultiplicative();
        while (this.scanner.currentSymbol() == Symbol.PLUS
                || this.scanner.currentSymbol() == Symbol.MINUS) {
            final boolean plus =
                this.scanner.currentSymbol() == Symbol.PLUS;
            this.scanner.next();
            final T v = this.parseMultiplicative();
            if (plus)
                value = this.create(value, (x, y) -> x + y, v);
            else
                value = this.create(value, (x, y) -> x - y, v);
        }
        return value;
    }

    public T parseMultiplicative() {
        T value = this.parseSimple();
```

```
        while (this.scanner.currentSymbol() == Symbol.TIMES
                || this.scanner.currentSymbol() == Symbol.DIV) {
            final boolean times =
                this.scanner.currentSymbol() == Symbol.TIMES;
            this.scanner.next();
            final T v = this.parseSimple();
            if (times)
                value = this.create(value, (x, y) -> x * y, v);
            else
                value = this.create(value, (x, y) -> x / y, v);
        }
        return value;
    }

    public T parseSimple() {
        if(this.scanner.currentSymbol() == Symbol.NUMBER) {
            return this.parseNumber();
        }
        if (this.scanner.currentSymbol() == Symbol.OPEN) {
            this.scanner.next();
            final T value = this.parseAdditive();
            if (this.scanner.currentSymbol() != Symbol.CLOSE)
                throw new RuntimeException(") expected");
            this.scanner.next();
            return value;
        }
        throw new RuntimeException("Number or ( expected");
    }

    public T parseNumber() {
        if (this.scanner.currentSymbol() != Symbol.NUMBER)
            throw new RuntimeException("Number expected");
        final double value = this.scanner.getNumber();
        this.scanner.next();
        return this.create(value);
    }

    protected abstract T create(
        final T left,
        final DoubleBinaryOperator op,
        final T right);

    protected abstract T create(
        final double value);
}
```

Diese abstrakte Klasse bildet nun die Grundlage für den `DoubleParserImpl` und den `ExpressionParserImpl` – diese abgeleiteten Klassen müssen nunmehr die `create`-Methoden von `AbstractParser` implementieren:

```
package parser;

import java.util.function.DoubleBinaryOperator;
import scanner.Scanner;

public class DoubleParserImpl extends AbstractParser<Double> {

    public DoubleParserImpl(final Scanner scanner) {
        super(scanner);
    }

    @Override
    protected Double create(
            final Double left,
            final DoubleBinaryOperator op,
            final Double right) {
        return op.applyAsDouble(left, right);
    }

    @Override
    protected Double create(final double value) {
        return value;
    }
}
```

```
package parser;

import java.util.function.DoubleBinaryOperator;
import expressions.BinaryExpression;
import expressions.Expression;
import expressions.NumberExpression;
import scanner.Scanner;

public class ExpressionParserImpl
        extends AbstractParser<Expression> {

    public ExpressionParserImpl(final Scanner scanner) {
        super(scanner);
    }

    @Override
    protected Expression create(
            final Expression left,
            final DoubleBinaryOperator op,
            final Expression right) {
        return new BinaryExpression(left, op, right);
    }

    @Override
    protected Expression create(final double value) {
        return new NumberExpression(value);
    }

}
```

Die Klassen `ValueParserTest` und `ExpressionParserTest` können unverändert übernommen werden. Diese Klassen testen nun natürlich auch den generischen `AbstractParser`...

3

Circuits

3 Circuits

Die Vorgaben für diese Übung und ein erster Einstieg befinden sich im Hauptscript. Sie sind im Kapitel 8, Abschnitt 2 beschrieben.

3.1 AbstractCircuit

In der Vorgabe war `Circuit` als Interface definiert. Wir können statt eines Interfaces eine abstrakte Basisklasse verwenden. Diese definiert zwei Attribute: `inputCount` und `outputCount` – und einen Konstruktor, der diese beiden Attribute initialisiert. Dann können die Interface-Methoden `getInputCount` und `getOutputCount` bereits in dieser abstrakten Klasse implementiert werden. Die instantiierbaren Klassen benötigen dann nur noch jeweils einen Konstruktor, der den Konstruktor der abstrakten Basisklasse aufruft.

Wir versuchen also, ein wenig zu refaktorieren:

```
package circuits;

public abstract class Circuit {

    private final int inputCount;
    private final int outputCount;

    public Circuit(final int inputCount, final int outputCount) {
        this.inputCount = inputCount;
        this.outputCount = outputCount;
    }

    public final int getInputCount() {
        return this.inputCount;
    }

    public final int getOutputCount() {
        return this.outputCount;
    }
}
```

Hier die instantiierbaren `Circuit`-Klassen:

```
package circuits;

public class AndCircuit extends Circuit {

    public AndCircuit() {
        super(2, 1);
    }
}
```

```
package circuits;

public class OrCircuit extends Circuit {

    public OrCircuit() {
        super(2, 1);
    }
}
```

```
package circuits;

public class NotCircuit extends Circuit {

    public NotCircuit() {
        super(2, 1);
    }
}
```

Die abgeleiteten Klassen sind merklich geschrumpft. Wir sind daher mit dem Ergebnis der Refaktorierung zufrieden. Die Testklasse war von den Änderungen nicht betroffen – wir können also dieselbe Testklasse wieder starten. Das Ergebnis ist wieder grün.

3.2 States

Die Eingänge und die Ausgänge einer jeden Schaltung haben einen Status: blau resp. rot (also: `false` resp. `true`).

Wir müssen den Status eines Eingangs resp. eines Ausgangs abfragen können. Wir entscheiden uns für mit einem Index parametrisierte Getter-Methoden, die auf jedes `Circuit` aufgerufen werden kann. Die Methoden bekommen den Namen `InputState` und `OutputState`.

Wir erweitern die Testklasse:

```
package test;
// ...
public class CircuitTest {
    @Test
    public void testAnd() {
        final Circuit c = new AndCircuit();
        Assert.assertEquals(2, c.getInputCount());
        Assert.assertFalse(c.inputState(0));
        Assert.assertFalse(c.inputState(1));
        Assert.assertFalse(c.outputState(0));
    }

    @Test
    public void testOr() {
        final Circuit c = new OrCircuit();
        Assert.assertEquals(2, c.getInputCount());
        Assert.assertFalse(c.inputState(0));
        Assert.assertFalse(c.inputState(1));
        Assert.assertFalse(c.outputState(0));
    }

    @Test
    public void testNot() {
        final Circuit c = new NotCircuit();
        Assert.assertEquals(1, c.getInputCount());
        Assert.assertFalse(c.inputState(0));
        Assert.assertTrue(c.outputState(0));
    }
}
```

Man beachte, die letzte Zeile der letzten Test-Methode: unmittelbar nach der Erzeugung eines `NotCircuit`-Objekts muss dessen Ausgang rot sein.

Wir müssen nun die Basisklasse `Circuit` erweitern.

Die Methoden `inputState` und `outputState` könnten zunächst einmal derart implementiert werden, dass der Compiler zufrieden ist und die Testklasse übersetzt: die Methoden könnten einfach `0` zurückliefern. Die letzte Testmethode würde dann zu Recht ein rotes Ergebnis liefern – weil der Ausgang `false` statt `true` wäre.

Wir überspringen diesen Schritt und bilden die beiden Methoden auf zwei Arrays ab: einen Array namens `inputStates` und einen weiteren Array namens `outputStates` (die Sichtbarkeit dieser Arrays ist `protected`: nur von `Circuit` abgeleitete Klassen sollen diese Arrays sehen und ansprechen können).

Im Konstruktor müssen dann die beiden Arrays erzeugt werden (erst dort ist die Größe der erforderlichen Arrays bekannt):

Hier die erweiterte `Circuit`-Klasse:

```
package circuits;

public abstract class Circuit {

    private final int inputCount;
    private final int outputCount;
    protected final boolean[] inputStates;
    protected final boolean[] outputStates;

    public Circuit(final int inputCount, final int outputCount) {
        this.inputCount = inputCount;
        this.outputCount = outputCount;
        this.inputStates = new boolean[inputCount];
        this.outputStates = new boolean[outputCount];
    }

    public final int getInputCount() {
        return this.inputCount;
    }

    public final int getOutputCount() {
        return this.outputCount;
    }

    public boolean inputState(final int index) {
        return this.inputStates[index];
    }

    public boolean outputState(final int index) {
        return this.outputStates[index];
    }
}
```

Warum entscheiden wird uns für Arrays statt für `Lists`? Wir wissen exakt, wie viele Elemente die jeweilige Datenstruktur besitzen muss. Wir würden also mit Kanonen auf Spatzen schießen, wenn wir `Lists` verwenden würden.

Warum prüfen wir in den Methoden `inputState` und `outputState` nicht die Gültigkeit des übergebenen Index? Weil diese Prüfung ohnehin beim Zugriff auf den Array erfolgt.

Leider liefert der Test immer noch rot – bei der letzten Testmethode.

Wir müssen also eine der abgeleiteten Klassen erweitern – die Klasse `NotCircuit`. Alle anderen Klassen bleiben unverändert:

```
package circuits;

public class NotCircuit extends Circuit {

    public NotCircuit() {
        super(1, 1);
        this.outputStates[0] = true;
    }
}
```

Der Testlauf liefert nun grün.

Ob der direkte Zugriff auf den `outputStates`-Array der Basisklasse sinnvoll ist, sei zunächst einmal dahingestellt. Wir behalten das Problem im Auge (machen uns eine entsprechende Notiz).

3.3 Refactoring

Ein Array kennt bekanntlich seine Größe. Die Methoden `getInputCount` und `getOutputCount` wurden bislang auf Attribute abgebildet. Diese sind nun unnötig geworden: die Methoden `getInputCount` und `getOutputCount` können einfach die Größe der `inputStates`- und `outputStates`-Arrays zurückliefern.

Wir bemerken als positiven Seiteneffekt, dass wir nun auch im Konstruktor auf zwei Zeilen verzichten können.

Hier die refaktorierte `Circuit`-Klasse:

```
package circuits;

public abstract class Circuit {

    protected final boolean[] inputStates;
    protected final boolean[] outputStates;

    public Circuit(final int inputCount, final int outputCount) {
        this.inputStates = new boolean[inputCount];
        this.outputStates = new boolean[outputCount];
    }

    public final int getInputCount() {
        return this.inputStates.length;
    }

    public final int getOutputCount() {
        return this.outputStates.length;
    }

    public boolean inputState(final int index) {
        return this.inputStates[index];
    }

    public boolean outputState(final int index) {
        return this.outputStates[index];
    }
}
```

Der Test liefert wieder grün. Wir können die Refaktorierung als gelungen betrachten.

3.4 Toggle

Nun versuchen wir uns am Togglen.

Nur Eingänge können getoggelt werden. Genauer: nur freie Eingänge können getoggelt werden. Aber bislang existieren ja noch keine Verbindungen, so dass alle Eingänge noch als frei betrachtet werden. Wir müssen also auf diese Preconditon später noch zurückkommen – wir behalten die Sache im Auge (wir haben eine entsprechende Notiz hinterlegt).

Wir fordern die Existenz einer Methode namens `toggle`. Diese muss auf jedes `Circuit` aufgerufen werden können. Wir werden die Methode daher in der Basisklasse spezifizieren müssen. Erst in den abgeleiteten Klassen kann sie implementiert werden – denn das Togglen hat bei einer And-Schaltung andere Auswirkungen als bei einer Or-Schaltung. Der Methode wird der Index des zu togglenden Eintrags übergeben.

Hier der erweiterte Test:

```
package test;
// ...
public class CircuitTest {

    // ...

    @Test
    public void testToggleAnd() {
        final Circuit c = new AndCircuit();
        c.toggle(0);
        Assert.assertTrue(c.inputState(0));
        Assert.assertFalse(c.inputState(1));
        Assert.assertFalse(c.outputState(0));
        c.toggle(1);
        Assert.assertTrue(c.inputState(0));
        Assert.assertTrue(c.inputState(1));
        Assert.assertTrue(c.outputState(0));
        c.toggle(0);
        Assert.assertFalse(c.inputState(0));
        Assert.assertTrue(c.inputState(1));
        Assert.assertFalse(c.outputState(0));
    }
    @Test
    public void testToggleOr() {
        final Circuit c = new OrCircuit();
        c.toggle(0);
        Assert.assertTrue(c.inputState(0));
        Assert.assertFalse(c.inputState(1));
        Assert.assertTrue(c.outputState(0));
        c.toggle(1);
        Assert.assertTrue(c.inputState(0));
        Assert.assertTrue(c.inputState(1));
        Assert.assertTrue(c.outputState(0));
        c.toggle(0);
        Assert.assertFalse(c.inputState(0));
        Assert.assertTrue(c.inputState(1));
        Assert.assertTrue(c.outputState(0));
```

```
        c.toggle(1);
        Assert.assertFalse(c.inputState(0));
        Assert.assertFalse(c.inputState(1));
        Assert.assertFalse(c.outputState(0));
    }
    @Test
    public void testToggleNot() {
        final Circuit c = new NotCircuit();
        c.toggle(0);
        Assert.assertTrue(c.inputState(0));
        Assert.assertFalse(c.outputState(0));
        c.toggle(0);
        Assert.assertFalse(c.inputState(0));
        Assert.assertTrue(c.outputState(0));
    }
}
```

Um den Compiler zufrieden zu stellen, könnten wir in der Basisklasse eine leere, nicht abstrakte `toggle`-Methode schreiben. Der Test würde dann natürlich rot. Wir entscheiden uns sofort für die "richtige" Lösung: wir deklarieren die Methode in der Basisklasse als `abstract` und implementieren sie in jeder der abgeleiteten Klassen:

```
package circuits;

public abstract class Circuit {
    // ...
    public abstract void toggle(final int index);
}
```

```
package circuits;

public class AndCircuit extends Circuit {
    // ...
    @Override
    public void toggle(final int index) {
        this.inputStates[index] = !this.inputStates[index];
        this.outputStates[0] = this.inputStates[0] &&
this.inputStates[1];
    }
}
```

```
package circuits;

public class OrCircuit extends Circuit {
    // ...
    @Override
    public void toggle(final int index) {
        this.inputStates[index] = !this.inputStates[index];
        this.outputStates[0] = this.inputStates[0] ||
this.inputStates[1];
    }
}
```

```
package circuits;

public class NotCircuit extends Circuit {
    // ...
    @Override
    public void toggle(final int index) {
        this.inputStates[index] = !this.inputStates[index];
        this.outputStates[0] = !this.inputStates[0];
    }
}
```

Jede `toggle`-Methode invertiert zunächst einmal den Status des entsprechenden Eingangs. Dann wird der Ausgang jeweils neu berechnet. Diese Berechnung ist natürlich jeweils unterschiedlich: eine And-Schaltung berechnet die and-Verknüpfung der Eingänge, eine Or-Schaltung die or-Verknüpfung ihrer Eingänge, und eine Not-Schaltung schließlich die Invertierung ihres Eingangs.

Der Test liefert grün.

3.5 Refactoring

Wir beobachten bei der obigen Lösung Code-Duplikation. In den `toggle`-Methoden der abgeleiteten Klassen findet sich immer die folgende Zeile:

```
this.inputStates[index] = !this.inputStates[index];
```

Wir können diese Zeile in die `toggle`-Methode der Basisklasse verschieben. Diese Methode kann dann natürlich nicht mehr abstrakt sein. Nachdem sie den Eingangsstatus invertiert hat, ruft sie eine neue Methode auf: `Calculate`. Und diese ist nun als `abstract` definiert und muss also in den abgeleiteten Klassen überschrieben werden. Die Methode hat die Aufgabe, aufgrund der aktuellen Eingangszustände die Zustände des Ausgangs (der Ausgänge) neu berechnen und zu setzen.

Nebenbei kann die `Calculate`-Methode dann auch bereits im Konstruktor der Basisklasse aufgerufen werden.

```
package circuits;

public abstract class Circuit {
    // ...

    protected abstract void calculate();

    public final void toggle(final int index) {
        this.inputStates[index] = !this.inputStates[index];
        this.calculate();
    }
}
```

```
package circuits;

public class AndCircuit extends Circuit {
    // ...
    @Override
    public void calculate() {
        this.outputStates[0] = this.inputStates[0] &&
this.inputStates[1];
    }
}
```

```
package circuits;

public class OrCircuit extends Circuit {
    // ...
    @Override
    public void calculate() {
        this.outputStates[0] = this.inputStates[0] ||
this.inputStates[1];
    }
}
```

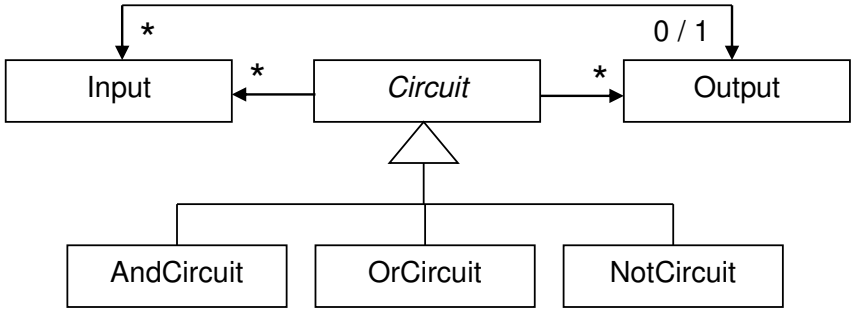
```
package circuits;

public class NotCircuit extends Circuit {
    // ...
    @Override
    public void calculate() {
        this.outputStates[0] = ! this.inputStates[0];
    }
}
```

Der Test liefert weiterhin grün. Wir können das Ergebnis der Refaktorierung als Erfolg verbuchen.

3.6 Input / Output

Bislang gab's nur `Circuits`.

In den kommenden Schritten werden wir Ausgänge mit Eingängen verbinden müssen. Dann spätestens werden wir um `Input`- und `Output`-Objekte nicht mehr umhinkommen. Ein `Output` kann mehrere `Inputs` versorgen – ein `Output` wird also eine Liste aller von ihm versorgten `Inputs` haben müssen. Ein `Input` wir möglicherweise eine Referenz erhalten, die auf das ihn versorgende `Output`-Objekt zeigt. Etc.



Insbesondere können dann natürlich die Stati, die bislang in boolschen Arrays der `Circuit`-Klasse gespeichert wurden, in den `Input`- resp. `Output`-Objekten gespeichert werden. Schließlich liegt es nahe, eine gemeinsame Oberklasse zu definieren, in der bereits das erforderliche Status-Attribut definiert ist. Der Name dieser Basisklasse sei `Port`.

Wir fangen klein an.

Wir werden zunächst einen kleinen Umbau vornehmen. Die Klasse `Circuit` wird einen Array von `Input`- und einen Array von `Output`-Objekten besitzen müssen. Dann sollte sie natürlich über Getter-Methoden verfügen, welche jeweils das index-te `Input`- resp. das index-te `Output`-Objekt zurückliefert: `input(i)` und `output(i)`. Der Status kann dann über die `Input`- resp. `Output`-Objekte ermittelt werden.

Der komplette Test muss entsprechend umgebaut werden. Der Status des i-ten Eingangs wird nun nicht mehr via `circuit.inputState(i)` ermittelt, sondern über den Aufruf von `circuit.input(0).getState()` (der Umbau kann aber rasch via find-replace geschehen):

```
package test;
// ...
public class CircuitTest {
    @Test
    public void testAnd() {
        final Circuit c = new AndCircuit();
        Assert.assertEquals(2, c.getInputCount());
        Assert.assertFalse(c.input(0).getState());
        Assert.assertFalse(c.input(1).getState());
```

```
        Assert.assertFalse(c.output(0).getState());
    }

    @Test
    public void testOr() {
        final Circuit c = new OrCircuit();
        Assert.assertEquals(2, c.getInputCount());
        Assert.assertFalse(c.input(0).getState());
        Assert.assertFalse(c.input(1).getState());
        Assert.assertFalse(c.output(0).getState());
    }

    @Test
    public void testNot() {
        final Circuit c = new NotCircuit();
        Assert.assertEquals(1, c.getInputCount());
        Assert.assertFalse(c.input(0).getState());
        Assert.assertTrue(c.output(0).getState());
    }

    @Test
    public void testToggleAnd() {
        final Circuit c = new AndCircuit();
        c.toggle(0);
        Assert.assertTrue(c.input(0).getState());
        Assert.assertFalse(c.input(1).getState());
        Assert.assertFalse(c.output(0).getState());
        c.toggle(1);
        Assert.assertTrue(c.input(0).getState());
        Assert.assertTrue(c.input(1).getState());
        Assert.assertTrue(c.output(0).getState());
        c.toggle(0);
        Assert.assertFalse(c.input(0).getState());
        Assert.assertTrue(c.input(1).getState());
        Assert.assertFalse(c.output(0).getState());
    }
    @Test
    public void testToggleOr() {
        final Circuit c = new OrCircuit();
        c.toggle(0);
        Assert.assertTrue(c.input(0).getState());
        Assert.assertFalse(c.input(1).getState());
        Assert.assertTrue(c.output(0).getState());
        c.toggle(1);
        Assert.assertTrue(c.input(0).getState());
        Assert.assertTrue(c.input(1).getState());
        Assert.assertTrue(c.output(0).getState());
        c.toggle(0);
        Assert.assertFalse(c.input(0).getState());
        Assert.assertTrue(c.input(1).getState());
        Assert.assertTrue(c.output(0).getState());
        c.toggle(1);
        Assert.assertFalse(c.input(0).getState());
        Assert.assertFalse(c.input(1).getState());
        Assert.assertFalse(c.output(0).getState());
    }
    @Test
    public void testToggleNot() {
        final Circuit c = new NotCircuit();
```

```
        c.toggle(0);
        Assert.assertTrue(c.input(0).getState());
        Assert.assertFalse(c.output(0).getState());
        c.toggle(0);
        Assert.assertFalse(c.input(0).getState());
        Assert.assertTrue(c.output(0).getState());
    }
}
```

In `Port` wird ein `state`-Attribute definiert. Die Getter-Methode für dieses Attribut ist `public`, die Setter-Methode besitzt die Package-Sichtbarkeit.

Die von `Port` abgeleiteten Klassen `Input` und `Output` sind noch leer. (Wir wissen aber bereits, dass es mit der Basisklasse allein nicht getan sein wird – ein `Input` unterscheidet sich funktional von einen `Output`. Und dieser Unterschied wird sich mit Sicherheit auch später in den Implementierungen der beiden Klassen niederschlagen.)

```
package circuits;

public class Port {
    private boolean state;
    void setState(final boolean state) {
        this.state = state;
    }
    public boolean getState() {
        return this.state;
    }
}
```

```
package circuits;

public class Input extends Port {
}
```

```
package circuits;

public class Output extends Port {
}
```

In der Klasse `Circuit` werden die boolschen Arrays `inputStates` und `outputStates` durch die Arrays `inputs` und `outputs` ersetzt. Diese werden im Konstruktor initialisiert – dort werden die entsprechenden `Input`- und `Output`-Objekte erzeugt und in die Arrays eingetragen. Schließlich werden die Methoden `inputState(i)` resp. `outputState(i)` ersetzt durch die Methode `input(i)` und `output(i)`, die jeweils das i-te `Input`- resp. das i-te `Output`-Objekt liefern.

```
package circuits;

public abstract class Circuit {

    protected final Input[] inputs;
    protected final Output[] outputs;

    public Circuit(final int inputCount, final int outputCount) {
        this.inputs = new Input[inputCount];
        this.outputs = new Output[outputCount];
        for (int i = 0; i < inputCount; i++)
            this.inputs[i] = new Input();
        for (int i = 0; i < outputCount; i++)
            this.outputs[i] = new Output();
    }

    protected abstract void calculate();

    public final void toggle(final int index) {
        this.inputs[index].setState(!this.inputs[index].getState());
        this.calculate();
    }

    public final int getInputCount() {
        return this.inputs.length;
    }

    public final int getOutputCount() {
        return this.outputs.length;
    }

    public Input input(final int index) {
        return this.inputs[index];
    }

    public Output output(final int index) {
        return this.outputs[index];
    }
}
```

Dann zwingt und der `Compiler` auch zu einer Änderung der abgeleiteten Klassen. Alle `caluclate`-Methoden müssen überarbeitet werden:

```
package circuits;

public class AndCircuit extends Circuit {
    // ...
    @Override
    public void calculate() {
        this.outputs[0].setState(
            this.inputs[0].getState()
            && this.inputs[1].getState());
    }
}
```

```
package circuits;

public class OrCircuit extends Circuit {
    // ...
    @Override
    public void calculate() {
        this.outputs[0].setState(
            this.inputs[0].getState()
            || this.inputs[1].getState());
    }
}
```

```
package circuits;

public class AndCircuit extends Circuit {
    // ...
    @Override
    public void calculate() {
        this.outputs[0].setState(
            ! this.inputs[0].getState());
    }
}
```

Der Test liefert weiterhin grün.

Zwar haben wir im letzten Schritt noch keine zusätzlich Funktionalität hinzugefügt – aber wird haben das System umgebaut und damit die Voraussetzungen für die folgenden Schritte bereitgestellt.

3.7 Refactoring

Bevor wir uns mit den Verbinden von `Outputs` und `Inputs` befassen, werfen wir noch einmal einen Blick auf die bisherige Lösung. Wir betrachten die `toggle`-Methode. Diese wurde bislang – mit einem Index versehen – auf `Circuits` aufgerufen. Wird ein `Circuit` getogglet oder nicht eher ein `Input`? Natürlich ein `Input`.

Also versuchen wir, die `toggle`-Methode von der `Circuit`-Klasse zur `Input`-Klasse zu verschieben. Dort benötigt sie natürlich auch keinen Index-Parameter – sie wird also parameterlos sein.

Wir reformulieren zunächst unsere Tests. Überall dort, wo bislang `circuit.toggle(i)` aufgerufen wurde, muss nun `circuit.input(i).toggle()` aufgerufen werden:

Wir passen zunächst den Test an:

```
package test;
// ...
public class CircuitTest {

    // ...

    @Test
    public void testToggleAnd() {
        final Circuit c = new AndCircuit();
        c.input(0).toggle();
        Assert.assertTrue(c.input(0).getState());
        Assert.assertFalse(c.input(1).getState());
        Assert.assertFalse(c.output(0).getState());
        c.input(1).toggle();
        Assert.assertTrue(c.input(0).getState());
        Assert.assertTrue(c.input(1).getState());
        Assert.assertTrue(c.output(0).getState());
        c.input(0).toggle();
        Assert.assertFalse(c.input(0).getState());
        Assert.assertTrue(c.input(1).getState());
        Assert.assertFalse(c.output(0).getState());
    }
    @Test
    public void testToggleOr() {
        // ...
    }
    @Test
    public void testToggleNot() {
        // ...
    }
}
```

Die `toggle`-Methode muss also zu einer Methode der Klasse `Input` werden.

Dabei taucht ein kleines Problem auf: Die bisherige `toggle`-Methode invertierte den Zustand des entsprechenden `Inputs` und rief dann die `Circuit`-Methode `calculate` auf. Auch die neue `toggle`-Methode muss dann natürlich `calculate` aufrufen. Das aber heißt, dass ein `Input` seinen Eigentümer – sein `Circuit` – kennen muss. Der Konstruktor von `Input` muss daher erweitert werden: ihm muss das `Circuit` übergeben werden. Dann sollte – aus Symmetrie-Gründen – auch ein `Output` seinen Owner kennen. Der Owner wird also zum Attribut der gemeinsamen Basisklasse `Port`:

```
package circuits;

public class Port {
    private final Circuit owner;
    private boolean state;
    public Port(final Circuit owner) {
        this.owner = owner;
    }
    public Circuit getOwner() {
        return this.owner;
    }
    void setState(final boolean state) {
        this.state = state;
    }
    public boolean getState() {
        return this.state;
    }
}
```

```
package circuits;

public class Input extends Port {
    public Input(final Circuit owner) {
        super (owner);
    }

    public void toggle() {
        this.setState(! this.getState());
        this.getOwner().calculate();
    }
}
```

```
package circuits;

public class Output extends Port {
    public Output(final Circuit owner) {
        super (owner);
    }
}
```

Schließlich zwingt uns der Compiler auch zu einer kleinen Änderung der `Circuit`-Klasse: Im Konstruktor dieser Klasse, wo die die `Input`-Objekte erzeugt werden. Dort muss nun dem Konstruktor von `Input` der `this`-Zeiger auf das aktuelle `Circuit` übergeben werden.

Und natürlich wird die `toggle`-Methode dieser Klasse entfernt.

```
package circuits;

public abstract class Circuit {

    protected final Input[] inputs;
    protected final Output[] outputs;

    public Circuit(final int inputCount, final int outputCount) {
        this.inputs = new Input[inputCount];
        this.outputs = new Output[outputCount];
        for (int i = 0; i < inputCount; i++)
            this.inputs[i] = new Input(this);
        for (int i = 0; i < outputCount; i++)
            this.outputs[i] = new Output(this);
    }

    abstract void calculate();

    // toggle wurde entfernt!

    // ...
}
```

Der Test liefert wieder grün.

Die Anwendung ist durch die Refaktorierung verständlicher geworden – wo bislang die `toggle`-Methode auf ein `Circuit` aufgerufen, wird sie nun auf ein `Input` eines solchen `Circuits` aufgerufen. Die neue Methode ist weniger komplex als die alte: sie ist parameterlos geworden.

Wir können also mit dem Ergebnis der Refaktorierung zufrieden sein.

3.8 Connect

Ausgänge sollen nun mit Eingängen verbunden werden können. Dabei muss der Eingang, mit dem ein Ausgang verbunden wird, natürlich auch dessen Status annehmen.

Wir spendieren der Klasse `Output` eine neue Methode: die `connect`-Methode. Als Parameter wird ihr derjenige `Input` übergeben, mit dem der aktuelle `Output` verbunden werden soll.

Wir testen zwei Fälle.

- Der (0-te) Ausgang einer And-Schaltung wird mit dem (0-ten) Eingang einer Not-Schaltung verbunden;
- Der Ausgang einer Not-Schaltung wird mit dem Eingang einer Not-Schaltung verbunden.

```
package test;
// ...
public class CircuitTest {

    // ...

    @Test
    public void testConnectAndWithNot()
    {
        final Circuit and = new AndCircuit();
        final Circuit not = new NotCircuit();

        and.output(0).connectTo(not.input(0));
        Assert.assertTrue(not.output(0).getState());
    }
    @Test
    public void testConnectNotWithNot()
    {
        final Circuit not0 = new NotCircuit();
        final Circuit not1 = new NotCircuit();

        not0.output(0).connectTo(not1.input(0));
        Assert.assertFalse(not1.output(0).getState());
    }

}
```

Die `connect`-Methode kann zunächst einmal leer implementiert werden (sie ist `void`). Beim Test werden wir dann rot sehen.

Wie wird `connect` nun "richtig" implementiert?

Die `Input`-Klasse wird um eine Referenz vom Typ `Output` erweitert – ein Attribut namens `source`. Diese Referenz ist entweder `null` oder zeigt auf dasjenige `Output`-Objekt, welches mit dem aktuellen Input verbunden ist.

Die Klasse `Output` ist um eine `List<Input>` erweitert worden. Diese Liste der `targets` enthält Referenzen auf diejenigen `Input`-Objekte, die von dem aktuellen `Output` versorgt werden.

```
package circuits;

public class Input extends Port {

    Output source;

    public Input(final Circuit owner) {
        super (owner);
    }

    public void toggle() {
        this.setState(! this.getState());
        this.getOwner().calculate();
    }
}
```

```
package circuits;
// ...
public class Output extends Port {

    final List<Input> targets = new ArrayList<>();

    public Output(final Circuit owner) {
        super(owner);
    }

    public void connectTo(final Input input) {
        input.source = this;
        this.targets.add(input);
        input.setState(this.getState());
        input.getOwner().calculate();
    }
}
```

Die `connect`-Methode trägt das aktuelle `Output`-Objekt in das `source`-Attribut des an `connect` übergebenen `Inputs` ein und fügt diesen `Input` zur `targets`-Liste hinzu. Dann wird der Status des `Output`-Objekts in das `Input`-Objekt übertragen und dessen Eigentümer veranlasst, die Stati seiner Ausgänge neu zu berechnen.

Der Test liefert grün.

3.9 Toggle When Connected

Werden beim Togglen eines `Inputs` die neu berechneten Ausgangs-Zustände korrekt entlang bestehender Verbinden "weitergereicht"?

Wir testen:

```
package test;
// ...
public class CircuitTest {

    // ...

    @Test
    public void testConnectAndWithNotAndToggle()
    {
        final Circuit and = new AndCircuit();
        final Circuit not = new NotCircuit();

        and.output(0).connectTo(not.input(0));

        and.input(0).toggle();
        Assert.assertTrue(not.output(0).getState());
        and.input(1).toggle();
        Assert.assertFalse(not.output(0).getState());
    }

}
```

Und sehen rot.

Wir erkennen, dass die Änderung eines `Input`-Zustands ganz andere Auswirkungen haben muss als die Änderung eines `Output`-Zustands. Bislang wurde das Zustands-Attribut in der Basisklasse `Port` definiert – dieses Attribut war das einzige Element der `Port`-Klasse. Wir können daher auf diese Basisklasse auch verzichten – sie bringt nicht viel – zumal `Inputs` und `Outputs` nirgendwo polymorph genutzt werden (es gibt keine Liste, die sowohl `Input`- als auch `Output`-Objekte enthält).

Wir vereinbaren in der `Input`-Klasse und der `Output`-Klasse also ein jeweils spezifisches `state`-Attribut– und zwar ein Attribut. Dann können wir in den für diese Attribute zuständigen Setter-Methoden jeweils spezifische Funktionalität unterbringen.

In der Klasse `Input` wird zunächst geprüft, ob der neu zu setzende Zustand bereits dem aktuelle Zustand gleicht – dann sind keine weiteren Aktionen erforderlich. Ansonsten wird der Zustand des `Inputs` geändert und anschließend die `calculate`-Methode des `Circuits` aufgerufen. Die `toggle`-Methode muss dann nur mehr den Zustand konvertieren und dabei die Setter-Methode für das `state`-Attribut aufrufen. Somit ist nun sichergestellt, dass immer dann, wenn der Zustand eines `Inputs` geändert wird, auch die `calculate`-Methode aufgerufen wird (und nicht nur dann, wenn getoggelt wird).

```
package circuits;

public class Input {

    private final Circuit owner;
    Output source;
    private boolean state;

    public Input(final Circuit owner) {
        this.owner = owner;
    }

    public Circuit getOwner() {
        return this.owner;
    }

    void setState(final boolean state) {
        if (this.state != state) {
            this.state = state;
            this.owner.calculate();
        }
    }

    public boolean getState() {
        return this.state;
    }

    public void toggle() {
        this.setState(!this.getState());
        this.getOwner().calculate();
    }
}
```

Auch in der Setter-Methode für das `state`-Attribut der `Output`-Klasse wird zunächst geprüft, ob überhaut etwas zu tun ist. Wenn ja, dann wird der Zustand neu gesetzt und an alle `targets` des `Outputs` propagiert – an alle mit dem `Output` verbundenen `Inputs`. Dabei wird jeweils die Setter-Methode für das `Input`-eigene `state`-Attribut aufgerufen.

Damit sollte dann das Problem des "Weiterreichens" systematisch gelöst worden sein:

```
package circuits;

import java.util.ArrayList;
import java.util.List;

public class Output {

    private boolean state;
    final List<Input> targets = new ArrayList<>();

    void setState(final boolean state) {
        if (state != this.state) {
            this.state = state;
            this.targets.forEach(target ->
target.setState(this.getState()));
        }
    }

    public boolean getState() {
        return this.state;
    }

    public void connectTo(final Input input) {
        input.source = this;
        this.targets.add(input);
        input.setState(this.getState());
        input.getOwner().calculate();
    }
}
```

Der Test liefert nun grün.

3.10 Preconditions

Bislang haben wir uns nicht weiter um Preconditions gekümmert – und die Tests waren so aufgebaut, dass Methoden niemals in einem Zustand aufgerufen wurden, in welchem sie eigentlich nicht hätten aufgerufen werden dürfen.

Wir schreiben nun einen Test, in welchem Methoden in "falschen" Zustand aufgerufen werden. Wir verlangen, dass in folgenden Situationen eine Exception geworfen wird:

- Auf ein `Input`, der bereits von einem `Output` versorgt wird, wird `toggle` aufgerufen.
- Ein `Input`, der bereits mit einem `Output` verbunden ist, wird mit einem weiteren `Output` verbunden.

Um in der Anwendung prüfen zu können, ob bestimmte Aktionen erlaubt sind oder nicht, muss die Anwendung ermitteln können, ob ein `Input` bereits von einem `Output` versorgt wird. Wir verlangen also in der Klasse `Input` eine Methode namens `isConnected`.

```
package test;
// ...
public class CircuitTest {

    // ...

    @Test
    public void testTogglePrecondition()
    {
        final Circuit and = new AndCircuit();
        final Circuit not = new NotCircuit();
        not.input(0).toggle();
        and.output(0).connectTo(not.input(0));
        XAssert.assertThrows(RuntimeException.class,
            () -> not.input(0).toggle());
    }

    @Test
    public void testConnectToPrecondition()
    {
        final Circuit and = new AndCircuit();
        final Circuit not = new NotCircuit();
        and.output(0).connectTo(not.input(0));
        XAssert.assertThrows(RuntimeException.class,
            () -> and.output(0).connectTo(not.input(0)));
    }

    @Test
    public void testConnected() {
        final Circuit and = new AndCircuit();
        final Circuit not = new NotCircuit();
        Assert.assertFalse(not.input(0).isConnected());
        and.output(0).connectTo(not.input(0));
        Assert.assertTrue(not.input(0).isConnected());
    }
}
```

Die `isConnected`-Methode könnte nun zunächst so implementiert werden, dass sie `false` liefert. Dann ist der Compiler bereits zufrieden. Der Test liefert rot.

Wie kommen wir von rot nach grün?

Die `isConnected`-Methode von `Input` liefert `true`, wenn die `source`-Property ungleich `null` ist – ansonsten `false`.

`toggle` wird um den Test einer Precondition erweitert: `source` muss `null` sein:

```
package circuits;

public class Input {

    // ...

    public void toggle() {
        if (this.isConnected())
            throw new RuntimeException("cannot toggle. input is
connected");
        this.setState(!this.getState());
        this.getOwner().calculate();
    }

    public boolean isConnected() {
        return this.source != null;
    }
}
```

Die `connectTo`-Methode wird ebenfalls um eine Precondition erweitert: Wenn der an `connectTo` übergebene `Input` nicht frei ist (also bereits versorgt ist), wirft die Methode eine Exception:

```
package circuits;
// ...
public class Output {

    // ...

    public void connectTo(final Input input) {
        if (input.isConnected())
            throw new RuntimeException(
                "cannot connectTo. input is connected");
        input.source = this;
        this.targets.add(input);
        input.setState(this.getState());
        input.getOwner().calculate();
    }
}
```

Der Test liefert nun grün.

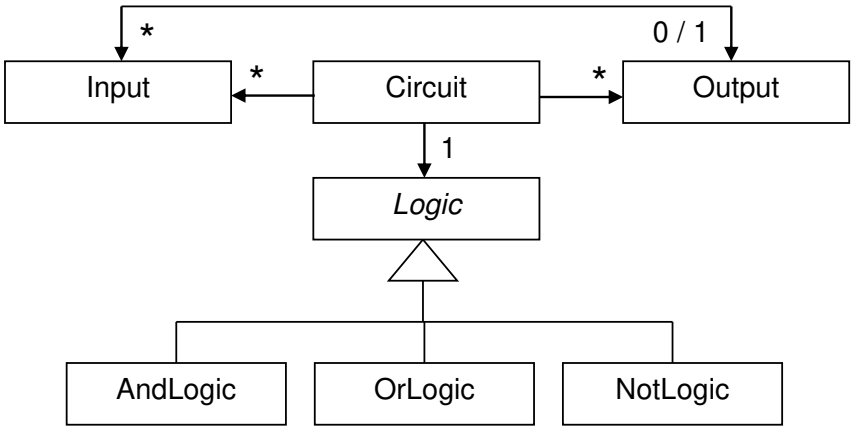
3.11 Logic

Wir gelangen nun zum letzten Use-Case: Der Typ einer bereits existierenden Schaltung soll geändert werden können – aus einer And-Schaltung z.B. sollte eine Or-Schaltung werden.

Ein Objekt kann zur Laufzeit bekanntlich seine Klasse nicht wechseln.

Ein `AndCircuit` unterscheidet sich von einem `OrCircuit` nur durch ihre unterschiedlichen Logiken – durch die unterschiedliche Implementierung von `Calculate`.

Wir könnten diese Berechnungslogik aus der `Circuit`-Klasse entfernen und in einem separaten Objekt unterbringen: in einem "Strategie"-Objekt. Wir definieren eine abstrakte `Logic`-Klasse und leiten von dieser Klasse `AndLogic`, `OrLogic` und `NotLogic` ab. Ein `Circuit` wird dann verbunden werden mit einem Objekt einer dieser von `Logic` abgeleiteten Klassen. Und das `Logic`-Objekt eines `Circuits` kann dann natürlich jederzeit durch ein `Logic`-Objekt eines anderen Typs ersetzt werden (vorausgesetzt, die Anzahl der Ein- und Ausgänge stimmen überein). Es sieht dann so aus, als habe das `Circuit` seine Klasse gewechselt.



Dann entfallen natürlich auch die von `Circuit` abgeleiteten Klassen: `Circuit` wird eine instanziierbar Klasse werden.

Wir wagen also einen kompletten Umbau der Anwendung. Zum Glück besitzen wir bereits eine ganze Reihe von Testmethoden, die nach dem Umbau wieder ausgeführt werden können…

Wir verlassen für einem Moment den `CircuitTest` und bauen eine weitere Testklasse: `LogicTest`. In dieser Klasse spezifizieren wird die Eigenschaften der erforderlichen `Logic`-Klassen:

```
package test;
// ...
public class LogicTest {
    @Test
    public void TestAndLogic() {
        final Logic logic = AndLogic.instance;
        Assert.assertEquals(2, logic.inputCount);
        Assert.assertEquals(1, logic.outputCount);
        Assert.assertTrue(logic.calc(new boolean[] { true, true
})[0]);
        Assert.assertFalse(logic.calc(new boolean[] { true, false
})[0]);
        Assert.assertFalse(logic.calc(new boolean[] { false, true
})[0]);
        Assert.assertFalse(logic.calc(new boolean[] { false,
false})[0]);
    }
    @Test
    public void TestOrLogic()
    {
        final Logic logic = OrLogic.instance;
        Assert.assertEquals(2, logic.inputCount);
        Assert.assertEquals(1, logic.outputCount);
        Assert.assertTrue(logic.calc(new boolean[] { true, true
})[0]);
        Assert.assertTrue(logic.calc(new boolean[] { true, false
})[0]);
        Assert.assertTrue(logic.calc(new boolean[] { false, true
})[0]);
        Assert.assertFalse(logic.calc(new boolean[] { false, false
})[0]);
    }
    @Test
    public void TestNotLogic()
    {
        final Logic logic = NotLogic.instance;
        Assert.assertEquals(1, logic.inputCount);
        Assert.assertEquals(1, logic.outputCount);
        Assert.assertFalse(logic.calc(new boolean[] { true })[0]);
        Assert.assertTrue(logic.calc(new boolean[] { false })[0]);
    }

}
```

Da `Logic`-Objekte zustandslos sind, können sie als Singletons implementiert werden. Wir verlangen, dass jede instanziierbare `Logic`-Klasse eins statisches `final`-Attribut namens `instance` hat, welches auf das einzige Objekt dieser Klasse zeigt.

Wir verlangen weiterhin, dass eine `Logic` Auskunft über die Anzahl der Ein- und Ausgänge geben können muss.

Und wir verlangen, dass auf jedes `Logic`-Objekt eine `calc`-Methode aufgerufen werden kann. Dieser `calc`-Methode wird ein boolscher Array übergeben; und sie liefert einen solchen boolschen Array zurück.

Die Basisklasse `Logic` definiert bereits die Methoden `getInputCount` und `getOutputCount`, Und sie spezifiziert die `calc`-Methode:

```
package logics;

public abstract class Logic {

    public final int inputCount;
    public final int outputCount;

    public Logic(final int inputCount, final int outputCount) {
        this.inputCount = inputCount;
        this.outputCount = outputCount;
    }

    public abstract boolean[] calc(boolean[] inputStates);
}
```

Jede abgeleitete Klasse definiert jeweils einen privaten Konstruktor, der den Konstruktor der Basisklasse aufruft (und dabei die Anzahl der Ein- und Ausgänge übergibt). Sie definiert eine statische `instance`-Variable (als `final`), die mit einer Instanz der Klasse initialisiert wird. Und sie implementiert jeweils die `calc`-Methode:

```
package logics;

public class AndLogic extends Logic {

    public static final AndLogic instance = new AndLogic();

    private AndLogic() {
        super(2, 1);
    }

    @Override
    public boolean[] calc(final boolean[] inputStates) {
        return new boolean[] { inputStates[0] && inputStates[1] };
    }
}
```

```
package logics;

public class OrLogic extends Logic {

    public static final OrLogic instance = new OrLogic();

    private OrLogic() {
        super(2, 1);
    }

    @Override
    public boolean[] calc(final boolean[] inputStates) {
        return new boolean[] { inputStates[0] || inputStates[1] };
    }
}
```

```
package logics;

public class NotLogic extends Logic {

    public static final NotLogic instance = new NotLogic();

    private NotLogic() {
        super(2, 1);
    }

    @Override
    public boolean[] calc(final boolean[] inputStates) {
        return new boolean[] { ! inputStates[0] };
    }
}
```

Der `LogicTest` liefert nun grün – und wir können uns wieder auf die "eigentliche" Baustelle begeben.

3.12 Circuits with Logic

Es wird keine `AndCircuits` mehr geben und auch keine `OrCircuits` und `NotCircuits`. Stattdessen wird es nun `Circuits` geben, die mit einer `AndLogic`, mit einer `OrLogic` oder mit einer `NotLogic` assoziiert sind.

Wir müssen also zunächst unseren Test an einigen Stellen ändern – nämlich überall an den Stellen, an denen `Circuit`-Objekte erzeugt werden. Die erforderlichen Änderungen können mit dem Find-Replace-Mechanismus ausgeführt werden:

```
new AndCircuit()    --> new Circuit(AndLogic.instance)
new OrCircuit()     --> new Circuit(OrLogic.instance)
new NotCircuit()    --> new Circuit(NotLogic.instance)
```

Hier die geänderte Testklasse, die zusätzlich um eine weitere Testmethode ergänzt wurde: `TestChangeAndToOr`.

```
package test;
// ...
public class CircuitTest {
    @Test
    public void testAnd() {
        final Circuit c = new Circuit(AndLogic.instance);
        // ...
    }

    @Test
    public void testOr() {
        final Circuit c = new Circuit(OrLogic.instance);
        // ...
    }

    @Test
    public void testNot() {
        final Circuit c = new Circuit(NotLogic.instance);
        // ...
    }

    // ...

    @Test
    public void testChangeAndToOr() {
        final Circuit c = new Circuit(AndLogic.instance);
        c.input(0).toggle();
        Assert.assertFalse(c.output(0).getState());
        c.setLogic(OrLogic.instance);
        Assert.assertTrue(c.output(0).getState());
    }
}
```

Die Circuit-Klasse wird nun eine instanziierbare Klasse. Dem Konstruktor wird eine `Logic` übergeben, die in der privaten Instanzvariablen `logic` gespeichert wird.

Es existiert eine Methode `setLogic`, welche die Neu-Kalkulation der Schaltung veranlasst.

Die `calculate`, die bislang abstrakt war, kann nun mühelos implementiert werden. Sie wird einfach auf den Aufruf der `calc`-Methode des mit dem `Circuit` assoziierten `Logic`-Objekts abgebildet.

```
package circuits;

import logics.Logic;

public final class Circuit {

    private Logic logic;
    protected final Input[] inputs;
    protected final Output[] outputs;

    public Circuit(final Logic logic) {
        this.logic = logic;
        this.inputs = new Input[logic.inputCount];
        this.outputs = new Output[logic.outputCount];
        for (int i = 0; i < logic.inputCount; i++)
            this.inputs[i] = new Input(this);
        for (int i = 0; i < logic.outputCount; i++)
            this.outputs[i] = new Output();
        this.calculate();
    }

    void calculate() {
        final boolean[] inputStates = new boolean[this.inputs.length];
        for (int i = 0; i < this.inputs.length; i++)
            inputStates[i] = this.inputs[i].getState();
        final boolean[] outputStates = this.logic.calc(inputStates);
        for (int i = 0; i < this.outputs.length; i++)
            this.outputs[i].setState(outputStates[i]);
    }

    public void setLogic(final Logic logic) {
        this.logic = logic;
        this.calculate();
    }

    public Logic getLogic() {
        return this.logic;
    }

    public int getInputCount() {
        return this.inputs.length;
    }

    public int getOutputCount() {
        return this.outputs.length;
    }
```

```
    public Input input(final int index) {
        return this.inputs[index];
    }

    public Output output(final int index) {
        return this.outputs[index];
    }
}
```

Der Test liefert grün.

Die `calculate`-Methode muss nun allerdings immer boolsche Arrays allokieren und initialisieren – und boolsche Arrays, die von `Calc` geliefert werden, wieder auslesen.

Natürlich hätten wir die `calc`-Methode derart spezifizieren können, dass ihr das jeweilige `Circuit` als Parameter übergeben worden wäre. Aber dann wäre die `Logic`-Klasse abhängig geworden von der Klasse `Circuit`. Natürlich ist es schöner, wenn diese Abhängigkeit nicht besteht – dann können die `Logic`-Klassen vielleicht noch in ganz anderen Anwendungen genutzt werden.

3.13 Circular Connections

Was passiert eigentlich, wenn ein `Output` der Schaltung A mit einem `Input` einer Schaltung B verbunden wird, und über B wiederum A "erreichbar" ist? Wenn also Schaltungen zirkulär verbunden werden?

Antwort: Eine solche Konstruktion sollte unter gar keinen Umständen erst möglich sein.

Wir verlangen also, dass `connectTo` eine Exception wirft, wenn das Resultat zirkulär wäre.

Dann sollte man – bevor `connectTo` aufgerufen wird – natürlich prüfen können, ob man von einem `Circuit` ein anderes "erreichen" kann. Wir verlangen also eine Methode namens `canReach`, die auf `Circuits` aufgerufen werden kann und der ein anderes `Circuit` als Parameter übergeben wird:

```
package test;
// ...
public class CircuitTest {

    // ...

    @Test
    public void TestCircularConnection()
    {
        final Circuit and = new Circuit(AndLogic.instance);
        final Circuit not = new Circuit(NotLogic.instance);
        Assert.assertFalse(and.canReach(not));
        and.output(0).connectTo(not.input(0));
        Assert.assertTrue(and.canReach(not));
        XAssert.assertThrows(RuntimeException.class,
            () -> not.output(0).connectTo(and.input(0)));
        XAssert.assertThrows(RuntimeException.class,
            () -> not.output(0).connectTo(and.input(1)));
    }
}
```

Wir implementieren `canReach` zunächst derart, dass `false` geliefert wird. Dann wird der Test scheitern. Wir sehen rot.

Wie gelangen wir von rot nach grün?

Zunächst wird die Klasse `Output` erweitert. Wir benötigen eine Methode `CanReach`:

```
package circuits;
// ...
public class Output {

    private final Circuit owner;

    private boolean state;
    final List<Input> targets = new ArrayList<>();

    public Output(final Circuit owner) {
        this.owner = owner;
    }

    public Circuit getOwner() {
        return this.owner;
    }

    void setState(final boolean state) {
        if (state != this.state) {
            this.state = state;
            this.targets.forEach(
                target -> target.setState(this.getState()));
        }
    }

    public boolean getState() {
        return this.state;
    }

    public void connectTo(final Input input) {
        if (input.isConnected())
            throw new RuntimeException(
                "cannot connectTo. input is connected");
        if (input.getOwner().canReach(this.getOwner()))
            throw new RuntimeException(
                "Circular connections not allowed");
        input.source = this;
        this.targets.add(input);
        input.setState(this.getState());
        input.getOwner().calculate();
    }

    public boolean canReach(final Circuit ciruit) {
        for (final Input input : this.targets) {
            if (input.getOwner() == ciruit
                    || input.getOwner().canReach(ciruit))
                return true;
        }
        return false;
    }
}
```

Die `canReach`-Methode ist rekursiv implementiert. Sie liefert `true`, wenn über den `Output` das als Parameter übergebene `Circuit` erreichbar ist (wie man sieht, wird hier nun die `owner`-Referenz tatsächlich auch benötigt).

Die `canReach`-Methode wird nun in `connectTo` aufgerufen – mit dem Ergebnis, dass `connectTo` eine Exception wirft, um eine zirkuläre Verbindung zu verhindern.

Weiterhin muss die im Test benutzte `canReach`-Methode implementiert werden.

```
package circuits;
// ...
public final class Circuit {

    // ...

    public boolean canReach(final Circuit other) {
        for (final Output output : this.outputs) {
            if (output.canReach(other))
                return true;
        }
        return false;
    }
}
```

Die Implementierung von `canReach` ist nun trivial – sie wird abgebildet auf die `canReach`-Methode der `Output`-Klasse.

Der Test liefert grün.

4

CSV-Dateien

4 CSV-Dateien

Die Vorgaben für diese Übung und ein erster Einstieg befinden sich im Hauptscript. Sie sind im Kapitel 8, Abschnitt 3 beschrieben.

4.1 Empty Lines

Der `CSVReader` sollte leere Zeilen überlesen.

Hier die Erweiterung der Test-Klasse:

```
package test;
// ...
public class CSVReaderTest {

    // ...

    @Test
    public void testNewLine() {

        final String s = "\n111;222\n\n333;444\n\n";

        final CSVReader reader = new CSVReader(
            new StringReader(s), 2, ";");
        Assert.assertArrayEquals(
            new String[] { "111", "222" },
            reader.readLine());
        Assert.assertArrayEquals(
            new String[] { "333", "444" },
            reader.readLine());

        Assert.assertNull(reader.readLine());
    }

}
```

Die Ausführung des Tests zeigt rot.

Die `readLine`-Methode von `CSVReader` wird wie folgt erweitert:

```
package util;
// ...
public class CSVReader {

    // ...

    public String[] readLine() {
        while (true) {
            try {
                final String line = this.reader.readLine();
                if (line == null)
                    return null;
                if (line.isEmpty())
                    continue;
                final String[] tokens = line.split(this.seperator);
                if (tokens.length != this.columnCount)
                    throw new RuntimeException(
                        "illegal line: " + line);
                return tokens;
            }
            catch (final IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

Die Ausführung des Tests zeigt grün.

4.2 Empty Lines

Zeilen, die führende oder abschließende Blanks enthalten, sollen "getrimmt" werden. Sofern dann eine leere Zeile übrigbleibt, soll diese überlesen werden.

Die Erweiterung der Testklasse:

```
package test;
//...
public class CSVReaderTest {

    //...

    @Test
    public void testEmptyLines() {
        final String s = "\n111;222\n\n   333;444\n    \n";
        final CSVReader reader =
            new CSVReader(new StringReader(s), 2, ";");
        Assert.assertArrayEquals(
            new String[] { "111", "222" },
            reader.readLine());
        Assert.assertArrayEquals(
            new String[] { "333", "444" },
            reader.readLine());
        Assert.assertNull(reader.readLine());
    }
}
```

Die Ausführung des Tests zeigt rot.

Die `readLine` von `CSVReader` wird erweitert:

```
package util;
// ...
public class CSVReader {

    // ...

    public String[] readLine() {
        while (true) {
            try {
                String line = this.reader.readLine();
                if (line == null)
                    return null;
                line = line.trim();
                if (line.isEmpty())
                    continue;
                final String[] tokens = line.split(this.seperator);
                if (tokens.length != this.columnCount)
                    throw new RuntimeException(
                        "illegal line: " + line);
                return tokens;
            }
            catch (final IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

Die Ausführung des Tests zeigt nun grün.

4.3 Trimmed Tokens

Auch die Tokens innerhalb einer Zeile, die führende oder abschließende Blanks enthalten, sollen "getrimmt" werden.

Die Erweiterung des Tests:

```
package test;
// ...
public class CSVReaderTest {

    // ...

    @Test
    public void testSurroundingWhitespace() {
        final String s = "\n  111;222  \n\n   333;\t444\n    \n";
        final CSVReader reader =
            new CSVReader(new StringReader(s), 2, ";");
        Assert.assertArrayEquals(
            new String[] { "111", "222" },
            reader.readLine());
        Assert.assertArrayEquals(
            new String[] { "333", "444" },
            reader.readLine());
        Assert.assertNull(reader.readLine());
    }
}
```

Die Ausführung des Tests zeigt rot.

Die `readLine` von `CSVReader` wird erweitert:

```
package util;
// ...
public class CSVReader {

    // ...

    public String[] readLine() {
        while (true) {
            try {
                String line = this.reader.readLine();
                if (line == null)
                    return null;
                line = line.trim();
                if (line.isEmpty())
                    continue;
                final String[] tokens = line.split(this.seperator);
                for (int i = 0; i < tokens.length; i++)
                    tokens[i] = tokens[i].trim();
                if (tokens.length != this.columnCount)
                    throw new RuntimeException(
                        "illegal line: " + line);
                return tokens;
            }
            catch (final IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

Die Ausführung des Tests zeigt nun grün.

4.4 Header-Zeile

Eine CSV-Eingabe soll eine Header-Zeile beinhalten können. Diese soll beliebig häufig abgefragt werden können.

Die Erweiterung der Test-Klasse:

```
package test;
// ...
public class CSVReaderTest {

    // ...

    @Test
    public void testHeader() {
        final String s =
            "isbn;title;year;price\n" +
            "1111;Pascal;1970;10.10\n" +
            "2222;Modula;1980;20.20\n";
        final CSVReader reader =
            new CSVReader(new StringReader(s), 4, ";", true);
        Assert.assertArrayEquals(
            new String[] { "isbn", "title", "year", "price" },
            reader.getHeader());
        Assert.assertArrayEquals(
            new String[] { "1111", "Pascal", "1970", "10.10" },
            reader.readLine());
        Assert.assertArrayEquals(
            new String[] { "2222", "Modula", "1980", "20.20" },
            reader.readLine());
        reader.getHeader()[0] = "Hello";
            // see clone in getHeader() !!
        Assert.assertArrayEquals(
            new String[] { "isbn", "title", "year", "price" },
            reader.getHeader());
        Assert.assertNull(reader.readLine());
    }
}
```

Damit der Compiler zufrieden ist, muss die Klasse `CSVReader` wird um einen zweiten Konstruktor und um eine getHeader-Methode erweitert werden:

```
package util;
// ...

public class CSVReader {

    public CSVReader(
            final Reader reader,
            final int columnCount,
            final String seperator,
            final boolean withHeader) {
    }

    public String[] getHeader() {
        return null;
    }

    // ...
}
```

Die Ausführung des Tests zeigt rot.

Die Klasse `CSVReader` wird wie folgt erweitert:

```
package util;
// ...
public class CSVReader {

    private final BufferedReader reader;
    private final int columnCount;
    private final String seperator;
    private final String[] header;

    public CSVReader(
            final Reader reader,
            final int columnCount,
            final String seperator,
            final boolean withHeader) {
        this.reader = new BufferedReader(reader);
        this.columnCount = columnCount;
        this.seperator = seperator;
        this.header = withHeader ? this.readLine() : null;
    }

    public CSVReader(
            final Reader reader,
            final int columnCount,
            final String seperator) {
        this(reader, columnCount, seperator, false);
    }

    public String[] getHeader() {
        if (this.header == null)
            throw new RuntimeException(
                "illegal call of getHeader");
        return this.header.clone();
```

```
    }

    public String[] readLine() {
        while (true) {
            try {
                String line = this.reader.readLine();
                if (line == null)
                    return null;
                line = line.trim();
                if (line.isEmpty())
                    continue;
                final String[] tokens = line.split(this.seperator);
                for (int i = 0; i < tokens.length; i++)
                    tokens[i] = tokens[i].trim();
                if (tokens.length != this.columnCount)
                    throw new RuntimeException(
                        "illegal line: " + line);
                return tokens;
            }
            catch (final IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

Die Ausführung des Tests zeigt nun grün.

4.5 Parser

Die Tokens einer Zeile werden bislang in Form von Strings geliefert. Solche Tokens sollten aber umgewandelt werden können in `Integer`-Werte, `Double`-Werte etc. (wir beschränken uns zunächst auf Wrapper-Objekte).

Hierzu benötigen wir ein Parser-Konzept. Wir verlassen die `CVSReader`-Baustelle und wenden uns der Parser-Baustelle zu.

Hier zunächst ein Test:

```
package test;
// ...
public class ParserSupportTest {
    @Test
    public void testStart() {
        Assert.assertEquals("Hello",
            ParserSupport.parse(String.class, "Hello"));
        Assert.assertEquals(42,
            ParserSupport.parse(Integer.class, "42"));
        Assert.assertEquals(3.14,
            ParserSupport.parse(Double.class, "3.14"), 0);
        Assert.assertTrue(
            ParserSupport.parse(Boolean.class, "true"));
        Assert.assertFalse(
            ParserSupport.parse(Boolean.class, "FAlse"));
        Assert.assertEquals('A',
            ParserSupport.parse(Character.class, "A"));
    }

    @Test
    public void testExceptions() {
        XAssert.assertThrows(RuntimeException.class,
            () -> ParserSupport.parse(Integer.class, "42abc"));
        XAssert.assertThrows(RuntimeException.class,
            () -> ParserSupport.parse(Character.class, "abc"));
        XAssert.assertThrows(RuntimeException.class,
            () -> ParserSupport.parse(Boolean.class, "wahr"));
    }
}
```

Wir setzen hier eine Klasse `ParserSupport` voraus, deren statische `parse`-Methode zwei Parameter übergeben werden: den Ziel-Typ der gewünschten Umwandlung (in Form eines `Class`-Objekts) und den umzuwandelnden String. Wird als Typ z.B. `int.class` angegeben, so soll `parse` auch einen `int`-Wert als Resultat liefern.

Wir benötigen ein Interface `Parser`:

```
package util;

@FunctionalInterface
public interface Parser<T> {
    public abstract T parse(final String s);
}
```

Und eine Klasse `ParserSupport`:

```
package util;
// ...
public class ParserSupport {

    public static <T> T parse(
            final Class<T> type,
            final String s) {
        return null;
    }
}
```

Der Compiler ist nun zufrieden.

Der Test zeigt natürlich rot an.

Hier nun die vollständige Implementierung von `ParserSupport`:

```
package util;
// ...
public class ParserSupport {

    private static final Map<Class<?>, Parser<?>> map
         = new HashMap<>();

    public static <T> void register(
            final Class<T> type,
            final Parser<T> parser) {
        Objects.requireNonNull(type);
        Objects.requireNonNull(parser);
        map.put(type, parser);
    }

    public static <T> T parse(
            final Class<T> type,
            final String s) {
        Objects.requireNonNull(type);
        Objects.requireNonNull(s);
        @SuppressWarnings("unchecked")
        final Parser<T> parser = (Parser<T>)map.get(type);
        return parser.parse(s);
    }

    static {
        register(String.class, s -> s);
        register(Integer.class, s -> Integer.parseInt(s));
        register(Long.class, s -> Long.parseLong(s));
        register(Float.class, s -> Float.parseFloat(s));
        register(Double.class, s -> Double.parseDouble(s));
        register(Boolean.class, s -> {
            if (s.equalsIgnoreCase("true"))
                return true;
            if (s.equalsIgnoreCase("false"))
                return false;
            throw new RuntimeException(
                "cannot convert '" + s + "' to boolean");
        });
        register(Character.class, s -> {
            if (s.length() != 1)
                throw new RuntimeException(
                    "cannot convert '" + s + "' to char");
            return s.charAt(0);
        });
    };
}
```

Der Test zeigt nun grün an.

4.6 ParserSupport für einfache Typen

Der `ParserSupport` funktioniert bislang nur für Wrapper-Typen, aber noch nicht für einfache Typen.

Der Test wird wie folgt erweitert:

```
package test;
// ...
public class ParserSupportTest {

    @Test
    public void testSimpleTypes() {
        Assert.assertEquals(42,
            ParserSupport.parse(int.class, "42"));
        Assert.assertEquals(3.14,
            ParserSupport.parse(double.class, "3.14"), 0);
        Assert.assertTrue(
            ParserSupport.parse(boolean.class, "true"));
        Assert.assertFalse(
            ParserSupport.parse(boolean.class, "FAlse"));
        Assert.assertEquals('A',
            ParserSupport.parse(char.class, "A"));
    }
}
```

Der Test liefert rot.

Wir führen eine Klasse `Primitives` ein:

```
package util;
// ...
public class Primitives {
    private Primitives() {
    }

    private static final Map<Class<?>, Class<?>> wrapperTypes
        = new HashMap<>();
    private static final Map<Class<?>, Class<?>> primitiveTypes
        = new HashMap<>();

    private static <T> void register(
            final Class<T> primitive,
            final Class<T> wrapper) {
        wrapperTypes.put(primitive, wrapper);
        primitiveTypes.put(wrapper, primitive);
    }

    static {
        register(byte.class, Byte.class);
        register(short.class, Short.class);
        register(int.class, Integer.class);
        register(long.class, Long.class);
        register(float.class, Float.class);
        register(double.class, Double.class);
        register(char.class, Character.class);
        register(boolean.class, Boolean.class);
    }
```

```
    @SuppressWarnings("unchecked")
    public static <T> Class<T> getPrimitive(
            final Class<T> wrapper) {
        return (Class<T>) primitiveTypes.get(wrapper);
    }

    @SuppressWarnings("unchecked")
    public static <T> Class<T> getWrapper(
            final Class<T> primitive) {
        return (Class<T>) wrapperTypes.get(primitive);
    }
}
```

Und können dann ParserSupport wie folgt erweitern:

```
package util;
// ...
public class ParserSupport {

    // ...

    public static <T> T parse(final Class<T> type, final String s) {
        Objects.requireNonNull(type);
        Objects.requireNonNull(s);
        final Class<?> lookupType =
            type.isPrimitive() ? Primitives.getWrapper(type) : type;
        @SuppressWarnings("unchecked")
        final Parser<T> parser = (Parser<T>) map.get(lookupType);
        return parser.parse(s);
    }

    // ...
}
```

Das Test-Resultat ist nun grün.

4.7 Mapper

Wir wenden uns einer neuen Baustelle zu: wir wollen String-Arrays auf Java-Objekte abbilden.

Wir verwenden folgende Demo-Klasse:

```
package test;

public class Book {

    public String isbn;
    public String title;
    public int year;
    public double price;

    public Book() {
    }

    public Book(
            final String isbn,
            final String title,
            final int year,
            final double price) {
        this.isbn = isbn;
        this.title = title;
        this.year = year;
        this.price = price;
    }

    @Override
    public int hashCode() { ... }

    @Override
    public boolean equals(final Object obj) { ... }

    @Override
    public String toString() { ... }
}
```

Die Test-Klasse:

```
package test;
// ...
public class MapperTest {

    @Test
    public void testStart() {
        final Mapper<Book> mapper = new Mapper<>(
            Book.class, "isbn", "title", "year", "price");
        final Book book = mapper.map("1111", "Pascal", "1970",
"10.10");
        Assert.assertEquals(
            new Book("1111", "Pascal", 1970, 10.10), book);
    }
    @Test
    public void testIllegalArgumentCount() {
        final Mapper<Book> mapper = new Mapper<>(
            Book.class, "isbn", "title", "year", "price");
        XAssert.assertThrows(RuntimeException.class,
            () -> mapper.map("1111", "Pascal", "1970"));
    }

    @Test
    public void testIllegalArgumentType() {
        final Mapper<Book> mapper = new Mapper<>(
            Book.class, "isbn", "title", "year", "price");
        XAssert.assertThrows(RuntimeException.class,
            () -> mapper.map("1111", "Pascal", "abc", "10.10"));
    }
}
```

Damit der Compiler zufrieden ist, benötigen wir folgende Klasse:

```
package util;
// ...
public class Mapper<T> {

    public Mapper(final Class<T> cls, final String... fieldNames) {
    }

    public T map(final String... stringValues) {
        return null;
    }
}
```

Der Test zeigt rot.

Hier die vollständige Implementierung der `Mapper`-Klasse:

```
package util;

import java.lang.reflect.Field;
import java.util.Objects;

public class Mapper<T> {

    private final Class<T> cls;
    private final String[] fieldNames;

    public Mapper(final Class<T> cls, final String... fieldNames) {
        Objects.requireNonNull(cls);
        Objects.requireNonNull(fieldNames);
        this.cls = cls;
        this.fieldNames = fieldNames;
    }

    public T map(final String... stringValues) {
        Objects.requireNonNull(stringValues);
        if (stringValues.length != this.fieldNames.length)
            throw new RuntimeException(
                "illegal count of stringValues");
        try {
            final T object = this.cls.newInstance();
            for (int i = 0; i < stringValues.length; i++) {
                final String fieldName = this.fieldNames[i];
                final String stringValue = stringValues[i];
                final Field field = this.cls.getField(fieldName);
                final Object value = ParserSupport.parse(
                    field.getType(), stringValue);
                field.set(object, value);
            }
            return object;
        }
        catch(final Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Der Test zeigt nun grün.

4.8 CSVMapper

Wie können nun einen CSVMapper schreiben, der die Funktionalitäten des CSVReaders und des Mappers miteinander verbindet.

Die Testklasse:

```
package test;
// ...
public class CSVMapperTest {

    @Test
    public void testWithHeader() {

        final String s =
            "isbn;title;year;price\n" +
            "1111;Pascal;1970;10.10\n" +
            "2222;Modula;1980;20.20\n";

        final CSVMapper<Book> mapper = new CSVMapper<>(
            Book.class, new StringReader(s), 4, ";");

        Assert.assertEquals(
            new Book("1111", "Pascal", 1970, 10.10),
            mapper.read());
        Assert.assertEquals(
            new Book("2222", "Modula", 1980, 20.20),
            mapper.read());
        Assert.assertNull(mapper.read());
    }
}
```

Wie stellen den Compiler zufrieden:

```
package util;
// ...
public class CSVMapper<T> {

    public CSVMapper(
            final Class<T> cls,
            final Reader reader,
            final int columnCount,
            final String seperator) {
    }

    public T read() {
        return null;
    }
}
```

Der Test zeigt rot.

Wir vervollständigen die Implementierung der Klasse `CSVMapper`:

```
package util;
// ...
public class CSVMapper<T> {

    private final CSVReader csvReader;
    private final Mapper<T> mapper;

    public CSVMapper(
            final Class<T> cls,
            final Reader reader,
            final int columnCount,
            final String seperator) {
        this.csvReader =
            new CSVReader(reader, columnCount, seperator, true);
        this.mapper =
            new Mapper<T>(cls, this.csvReader.getHeader());
    }

    public T read() {
        final String[] line = this.csvReader.readLine();
        if (line == null)
            return null;
        return this.mapper.map(line);
    }
}
```

Der Test zeigt grün.

4.9 CSVMapper ohne Header

Der `CSVMapper` sollte auch ohne Header-Zeile arbeiten können. Dann muss dem Konstruktor der Array der Namen übergeben werden:

Wir erweitern die Testklasse:

```
package test;
// ...
public class CSVMapperTest {

    @Test
    public void testWithHeader() {
        // ...
    }

    @Test
    public void testWithoutHeader() {
        final String s =
            "1111;Pascal;1970;10.10\n" +
            "2222;Modula;1980;20.20\n";
        final String[] header =
            new String[] { "isbn", "title", "year", "price" };
        final CSVMapper<Book> mapper = new CSVMapper<>(
            Book.class, new StringReader(s), 4, ";", header);
        Assert.assertEquals(
            new Book("1111", "Pascal", 1970, 10.10),
            mapper.read());
        Assert.assertEquals(
            new Book("2222", "Modula", 1980, 20.20),
            mapper.read());
        Assert.assertNull(mapper.read());
    }
}
```

Die Klasse `CSVMapper` wird um einen zweiten Konstruktor erweitert:

```
package util;
// ...
public class CSVMapper<T> {

    // ...

    public CSVMapper(
            final Class<T> cls,
            final Reader reader,
            final int columnCount,
            final String seperator,
            final String[] header) {
    }
}
```

Der Test zeigt rot.

Wir vervollständigen die Implementierung:

```
package util;
// ...
public class CSVMapper<T> {

    private final CSVReader csvReader;
    private final Mapper<T> mapper;

    public CSVMapper(
            final Class<T> cls,
            final Reader reader,
            final int columnCount,
            final String seperator) {
        this.csvReader =
            new CSVReader(reader, columnCount, seperator, true);
        this.mapper =
            new Mapper<T>(cls, this.csvReader.getHeader());
    }

    public CSVMapper(
            final Class<T> cls,
            final Reader reader,
            final int columnCount,
            final String seperator,
            final String[] header) {
        this.csvReader = new CSVReader(reader, columnCount,
seperator);
        this.mapper = new Mapper<T>(cls, header);
    }

    public T read() {
        final String[] line = this.csvReader.readLine();
        if (line == null)
            return null;
        return this.mapper.map(line);
    }
}
```

Der Test zeigt nun grün.

4.10 Refactoring

Die `CSVMapper`-Klasse sollte ein wenig refaktoriert werden: der zweite Konstruktor kann auf den ersten Konstruktor zurückgeführt werden.

Hier das Resultat:

```
package util;
// ...
public class CSVMapper<T> {

    private final CSVReader csvReader;
    private final Mapper<T> mapper;

    public CSVMapper(
            final Class<T> cls,
            final Reader reader,
            final int columnCount,
            final String seperator,
            final String[] header) {
        this.csvReader = new CSVReader(
            reader, columnCount, seperator, header == null);
        this.mapper = new Mapper<T>(
            cls, header == null ? this.csvReader.getHeader() :
header);
    }

    public CSVMapper(
            final Class<T> cls,
            final Reader reader,
            final int columnCount,
            final String seperator) {
        this(cls, reader, columnCount, seperator, null);
    }

    public T read() {
        final String[] line = this.csvReader.readLine();
        if (line == null)
            return null;
        return this.mapper.map(line);
    }
}
```

Der Test zeigt weiterhin grün…

4.11 Mapping aller CSV-Zeilen

Der `CSVMapper` sollte um eine Methode erweitert werden, welche alle CSV-Zeilen einliest, sie auf Java-Objekt abbildet und eine Liste dieser Objekte zurückliefert.

Wir erweitern `CSVMapperTest`:

```
package test;
// ...
public class CSVMapperTest {

    // ...

    @Test
    public void testMapAll() {
        final String s =
            "isbn;title;year;price\n" +
            "1111;Pascal;1970;10.10\n" +
            "2222;Modula;1980;20.20\n";
        final CSVMapper<Book> mapper = new CSVMapper<>(
            Book.class, new StringReader(s), 4, ";");
        final List<Book> books = mapper.readAll();
        Assert.assertEquals(
            new Book("1111", "Pascal", 1970, 10.10),
            books.get(0));
        Assert.assertEquals(
            new Book("2222", "Modula", 1980, 20.20),
            books.get(1));
    }
}
```

Hier die erforderliche Erweiterung der `CSVMapper`-Klasse:

```
package util;
// ...
public class CSVMapper<T> {
    // ...
    public List<T> readAll() {
        final List<T> list = new ArrayList<>();
        for (T obj = this.read(); obj != null; obj = this.read())
            list.add(obj);
        return list;
    }
}
```

Der Test zeigt grün.

4.12 Default-Values

Sofern ein Token in der Eingabe leer ist, sollte ein Default-Wert verwendet werden können.

Wir erweitern die Demo-Klasse Book:

```
package test;

import util.Default;

public class Book {

    public String isbn;

    @Default("N.N.")
    public String title;

    public int year;

    @Default("0")
    public double price;

    // ...
}
```

Wir benötigen zunächst eine `@Default`-Annotation:

```
package util;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Default {
    public abstract String value();
}
```

Wir erweitern den `MapperTest`:

```
package test;
// ...
public class MapperTest {

    // ...

    @Test
    public void testDefault() {
        final Mapper<Book> mapper = new Mapper<>(
            Book.class, "isbn", "title", "year", "price");
        final Book book = mapper.map("1111", "", "1970", "");
        Assert.assertEquals(new Book("1111", "N.N.", 1970, 0.0),
book);
    }
}
```

Der Test zeigt rot.

Wir erweitern schließlich die `Mapper`-Implementierung:

```
package util;
// ...
public class Mapper<T> {

    private final Class<T> cls;
    private final String[] fieldNames;

    public Mapper(final Class<T> cls, final String... fieldNames) {
        Objects.requireNonNull(cls);
        Objects.requireNonNull(fieldNames);
        this.cls = cls;
        this.fieldNames = fieldNames;
    }

    public T map(final String... stringValues) {
        Objects.requireNonNull(stringValues);
        if (stringValues.length != this.fieldNames.length)
            throw new RuntimeException(
                "illegal count of stringValues");
        try {
            final T object = this.cls.newInstance();
            for (int i = 0; i < stringValues.length; i++) {
                final String fieldName = this.fieldNames[i];
                String stringValue = stringValues[i];
                final Field field = this.cls.getField(fieldName);
                if (stringValue.isEmpty()) {
                    final Default d =
                        field.getAnnotation(Default.class);
                    if (d != null)
                        stringValue = d.value();
                }
                final Object value = ParserSupport.parse(
                    field.getType(), stringValue);
                field.set(object, value);
            }
            return object;
        }
        catch(final Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Und der Test zeigt grün.

Und es schadet natürlich nicht, alle Tests noch einmal zusammenzufassen:

```
package test;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    CSVReaderTest.class,
    MapperTest.class,
    CSVMapperTest.class
})

public class AlltogetherTest {
}
```