Lektion 18

Streams (Character Streams)

Unit Tests II

Standardeingabe und –ausgabe

java.lang

Class System

java.lang.Object
    java.lang.System

```
public final class System
extends Object
```

The System class contains several useful class fields and methods. It cannot be instantiated.

Among the facilities provided by the System class are standard input, standard output, and error output streams; access to externally defined properties and environment variables; a means of loading files and libraries; and a utility method for quickly copying a portion of an array.

Since:

JDK1.0

Field Summary

Fields

Modifier and Type	Field and Description
static PrintStream	**err** The "standard" error output stream.
static InputStream	**in** The "standard" input stream.
static PrintStream	**out** The "standard" output stream.

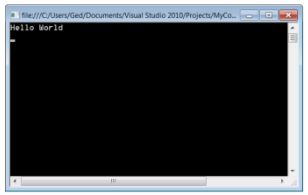
System.in und System.out

java.io

<<abstract>>
OutputStream

FilterOutputStream

PrintStream



```
public static void main(String[] args)
    throws IOException
{
    PrintStream out = System.out;
    InputStream in = System.in;
    int i;
    do {
        i = in.read();
        if (i != -1) out.write(i);
    }
    while (i != -1);
}
```

<<abstract>>
InputStream

FilterInputStream

BufferedInputStream

Speicher

ByteArrayOutputStream und ByteArrayInputStream

- Mit einem ByteArrayOutputStream kann in den JVM Speicher geschrieben werden.

- Mit einem ByteArrayInputStream kann ein byte[] in einen Input-Stream verwandelt werden.

```
public class MethodeZumDateiLesen {
  static void copyImproved(InputStream is, OutputStream os) throws IOException {
    byte[] b = new byte[4096];
    int n;
    do {
      n = is.read(b);
      if (n != -1) os.write(b, 0, n);
    }
    while (n != -1);
  }
  public static String readFromFile(String filename) throws IOException {
    FileInputStream fis = new FileInputStream(filename);
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    copyImproved(fis, baos);
    fis.close();
    return baos.toString();
  }

  public static void main(String[] args) throws IOException {
    String s = readFromFile("hello.txt");
    System.out.println(s);
  }
```

Kein Netzwerkhost und -port, keine Datei in die geschrieben wird…
ByteArrayOutputStream schreibt in den Speicher des JVM Prozesses.

Alles, was bisher in den Stream geschrieben wurde,
wird in einen String verwandelt.

Character Streams

Bisher mussten wir immer mit Bytes hantieren.

Das ist notwendig, wenn wir mit Binärdaten arbeiten.

Das ist umständlich, wenn wir Texte versenden wollen.

```
public class TCPClient
{
  public static void main(String[] args)
  {
    final int PORT = 5000;
    final String HOST = "localhost";
    try (Socket connectionToServer = new Socket(HOST, PORT);
         OutputStream os = connectionToServer.getOutputStream();)
    {
      os.write("Suppe mit dem Löffel löffeln\n".getBytes());
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

Verbindung zum Server herstellen

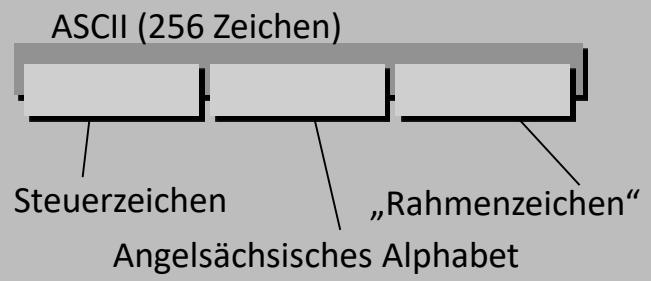
OutputStream holen, um zum Server zu schreiben

Durch getBytes() wird der String in ein Byte-Array konvertiert.
Doch wie erfolgt die Kodierung?

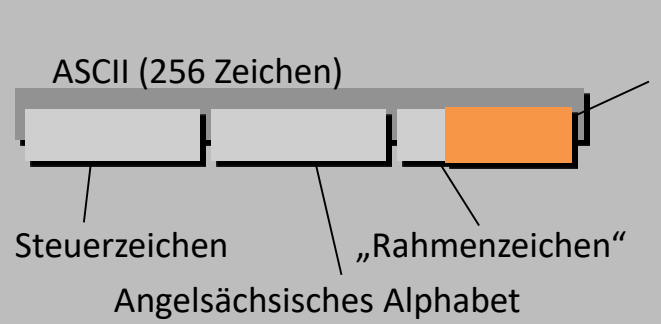
Kodierung von Zeichen

Wie werden Zeichen und Zeichenketten in Bytefolgen umgewandelt?

ASCII-Code: Nur angelsächsische Zeichen



Modifikation durch Code Pages



Landesspezifische Sonderzeichen gem. Code Page:

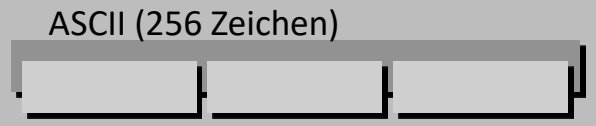
819: Latin-1
1252: Westeuropäisch
…

Um eine Bytefolge als Zeichenkette interpretieren zu können, ist es notwendig, die zugrundeliegende Kodierungstabelle (z.B. im Betriebssystem) festzulegen.

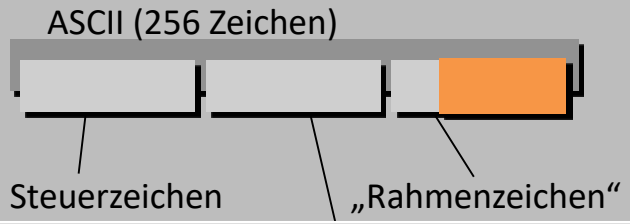
Kodierung von Zeichen

Wie werden Zeichen und Zeichenketten in Bytefolgen umgewandelt?

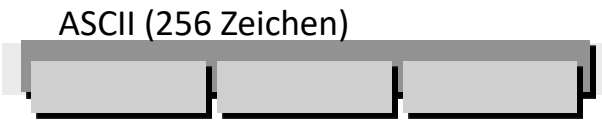
ASCII-Code: Nur angelsächsische Zeichen



Modifikation durch Code Pages



Unicode



Unicode 3.0:
(<65.536 Zeichen – 2 Byte)
September 1999

Seit Unicode 3.1:
(<4.294.967.296 Zeichen – 4 Byte)
März 2001

Kodierungen: UTF-8: min. 1 Byte pro Zeichen, max. 4; UTF-16: min. 2 Bytes, max. 4; UTF-32: 4 Bytes pro Zeichen.

```
public class TCPClient
{
  public static void main(String[] args)
  {
    final int PORT = 5000;
    final String HOST = "localhost";
    try (Socket connectionToServer = new Socket(HOST, PORT);
         OutputStream os = connectionToServer.getOutputStream();)
    {
      os.write("Suppe mit dem Löffel löffeln\n".getBytes()); //Charset.defaultCharset()
      os.write("Suppe mit dem Löffel löffeln\n".getBytes(StandardCharsets.ISO_8859_1));
      os.write("Suppe mit dem Löffel löffeln\n".getBytes(StandardCharsets.UTF_16));
      os.write("Suppe mit dem Löffel löffeln\n".getBytes(StandardCharsets.UTF_8));
      os.write("Suppe mit dem Löffel löffeln\n".getBytes(StandardCharsets.US_ASCII));
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

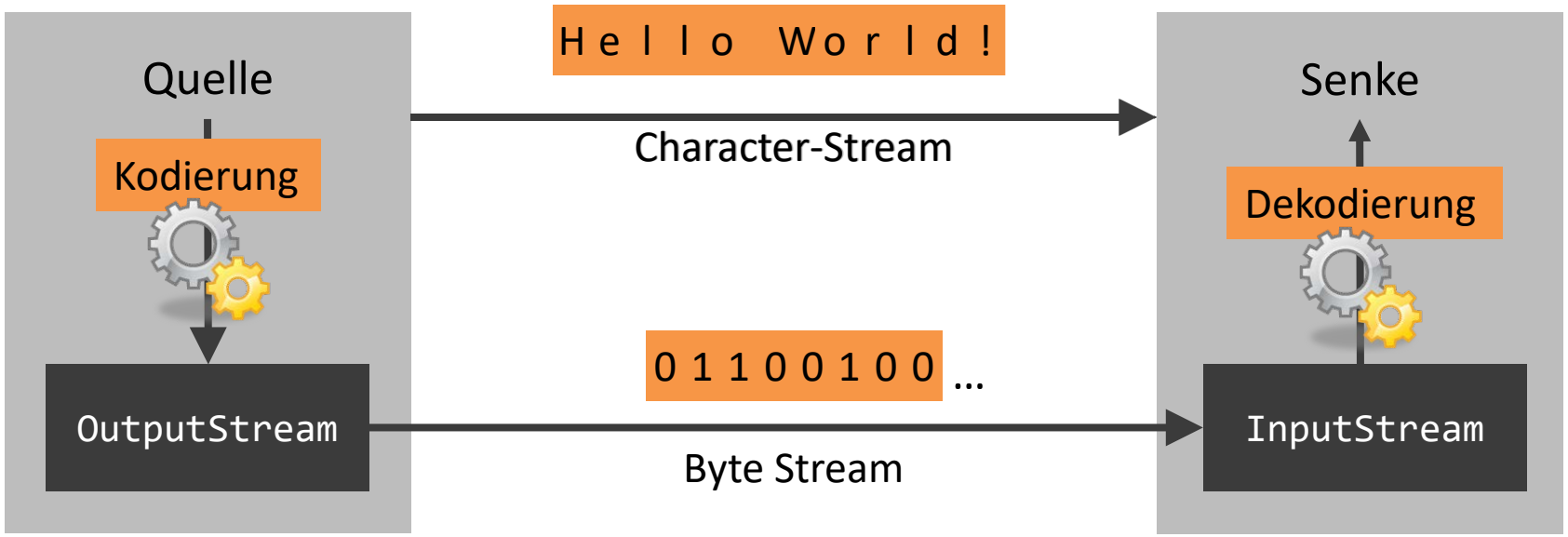
Es gibt viele Möglichkeiten, den Satz zu übertragen.
Der Entwickler muss wissen, welche Kodierung der Server versteht.

Müssen wir jedes Mal, wenn wir einen String schreiben, den String in Bytes verwandeln?

Es gibt Character Streams, die uns diese Arbeit abnehmen.

Character-Streams

Um das Senden und Empfangen von Zeichen und Zeichenketten zu vereinfachen, unterstützt Java Character-Streams.

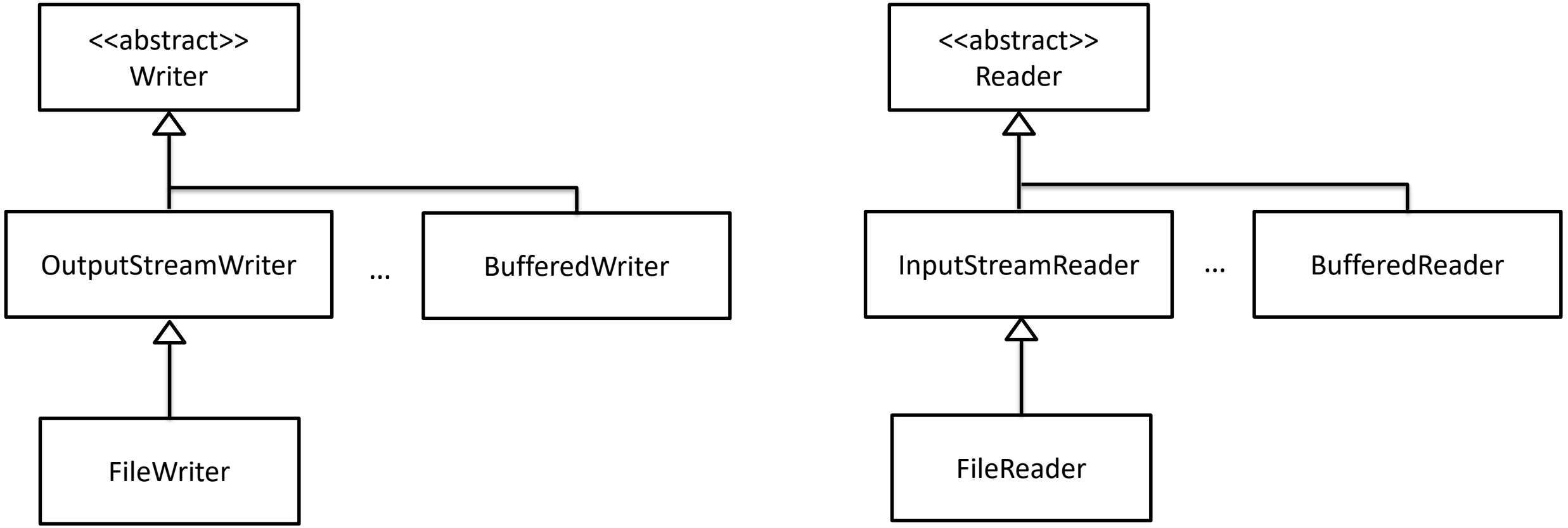


Character-Streams


```
public abstract class Reader
  implements Readable, Closeable {

  abstract public int read(char[] c, int off, int len)
    throws IOException;
  public int read() throws IOException {…}
  public int read(char[] c) throws IOException {…}

  public boolean ready() throws IOException {…}

  abstract public void close() throws IOException;
}
```

```
public abstract class Writer
  implements Appendable, Closeable, Flushable {

  abstract public void write(char[] c, int off, int len)
    throws IOException;
  public void write(int c) throws IOException {…}
  public void write(char[] c) throws IOException {…}
  public void write(String str) throws IOException {…}
  public void write(String str, int off, int len)
    throws IOException {…}

  abstract public void flush() throws IOException;
  abstract public void close() throws IOException;
}
```

Textuelle Daten werden durch sog. **Character Streams** verarbeitet.

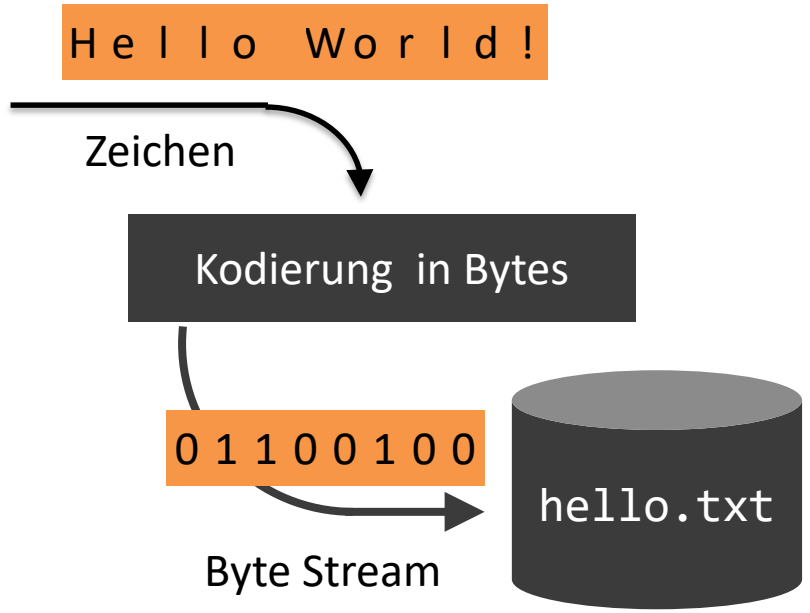


Beim Erstellen des Character Streams kann im Konstruktor das Charset mitangegeben werden.
(UTF-8, UTF-16, …)

OutputStreamWriter

Im nachfolgenden Beispiel wird eine Zeichenkette mittels `OutputStreamWriter` kodiert und über einen `FileOutputStream` in eine Datei geschrieben.

```
public static void main(String[] args)
{
  try(FileOutputStream fos = new FileOutputStream("hello.txt");
      OutputStreamWriter osw = new OutputStreamWriter(fos);)
  {
    osw.write("Hello World!");
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```



OutputStreamWriter ist eine Art “Adapter”, der aus einem (Byte) Stream, einen Writer (Character Stream) macht.

Das Verschachteln von OutputStream und OutputStreamWriter ist zu umständlich?
Für diese häufige Kombination gibt es die **FileWriter**-Klasse als Convenience-Klasse.

OutputStreamWriter(FileOutputStream)

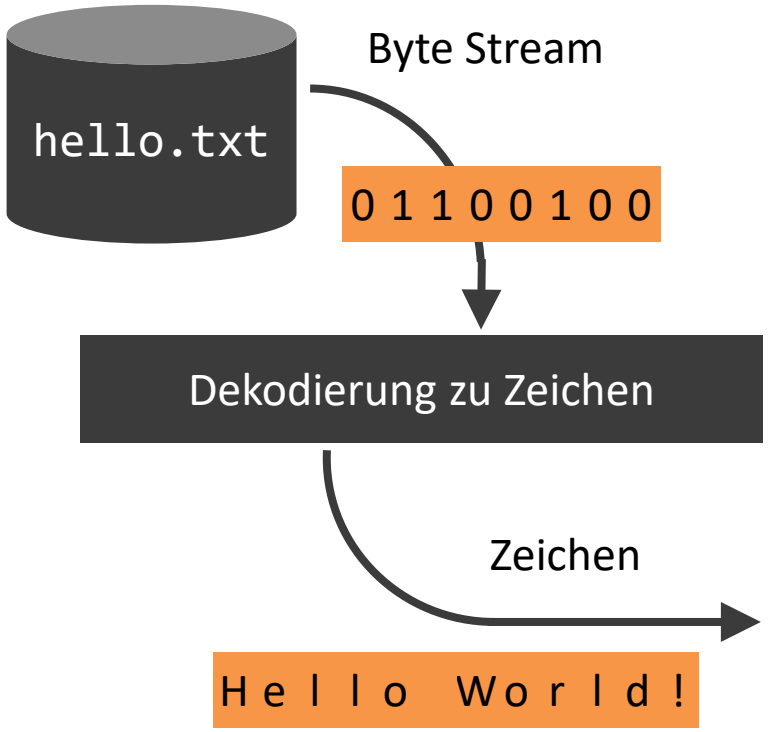
```
public static void main(String[] args)
{
  try(FileOutputStream fos = new FileOutputStream("h.txt");
      OutputStreamWriter osw = new OutputStreamWriter(fos);)
  {
    osw.write("Hello World!");
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

FileWriter

```
public static void main(String[] args)
{
  try(FileWriter fw = new FileWriter("h.txt");)
  {
    fw.write("Hello World!");
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

InputStreamReader

Die eingelesenen Zeichen werden in einem char-Array abgelegt.

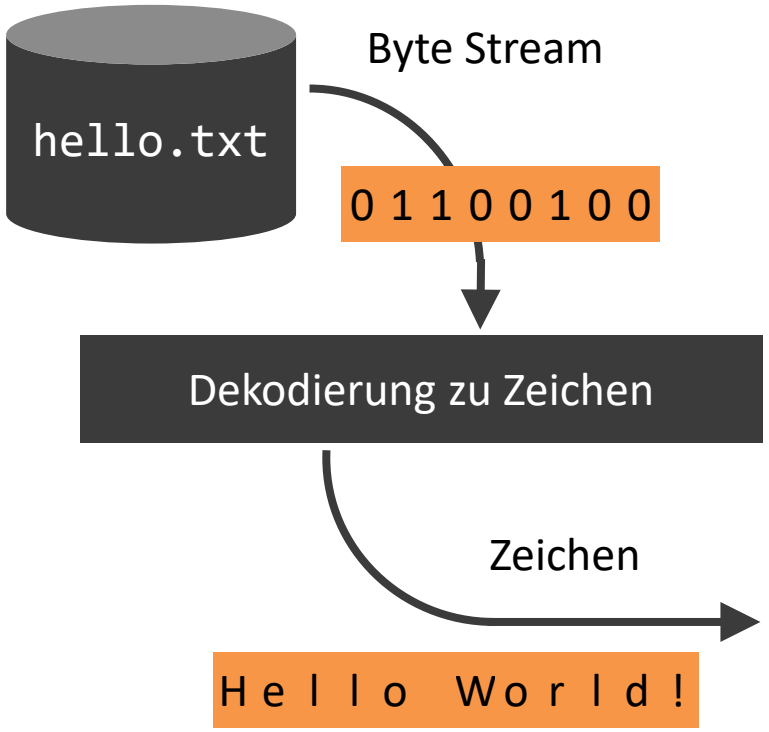



```
public static void main(String[] args)
{
  try(FileInputStream fis = new FileInputStream("hello.txt");
      InputStreamReader isr = new InputStreamReader(fis);)
  {
    char[] data = new char[20];
    int n = -1;
    while (true)
    {
      n = isr.read(data);
      if (n == -1) break;
      String s = new String(data, 0, n);
      System.out.println(s);
    }
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

InputStreamReader ist eine Art “Adapter”, der aus einem (Byte) Stream, einen Reader (Character Stream) macht.

FileReader

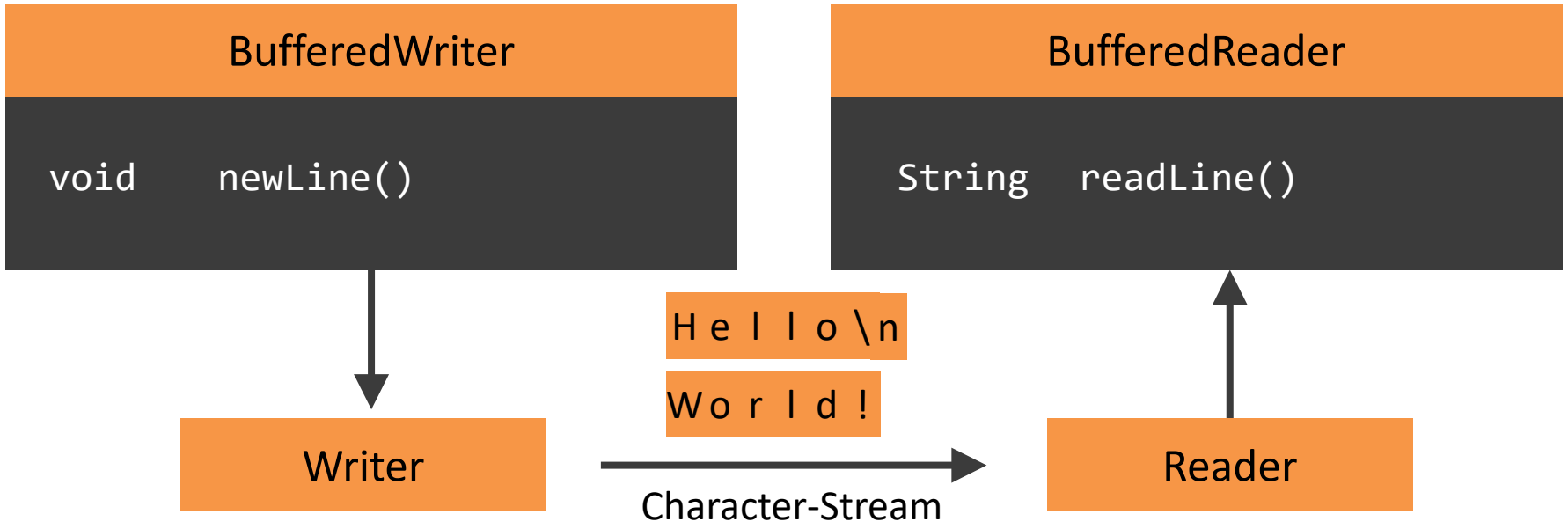
Auch beim Lesen von Textdateien gibt es eine Convenience-Klasse




```
public static void main(String[] args)
 {
   try(FileReader fr = new FileReader("hello.txt");)
   {
     char[] data = new char[20];
     int n = -1;
     while (true)
     {
       n = fr.read(data);
       if (n == -1) break;
       String s = new String(data, 0, n);
       System.out.println(s);
     }
   }
   catch (Exception e)
   {
     e.printStackTrace();
   }
 }
```

BufferedWriter und BufferedReader

Durch die Nutzung von `BufferedWriter` und `BufferedReader` wird ein Character-Stream durch Zeilen strukturiert. Bei der Quelle können Zeilenumbrüche z.B. mit `newLine()` eingefügt, bei der Senke Strings zeilenweise mit `readLine()` gelesen werden.



BufferedReader

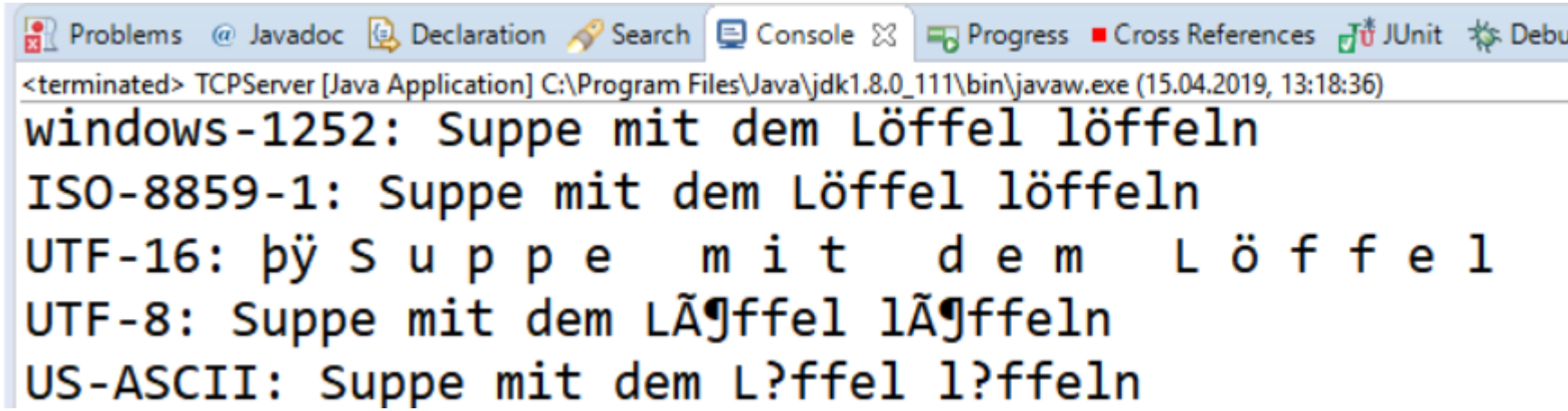
Zeilenweises Lesen mit BufferedReader

```
public static void main(String[] args)
{
  try(BufferedReader br = new BufferedReader(new FileReader("hello.txt"));)
  {
    do
    {
      String line = br.readLine();
      if (line == null) break;
      System.out.println(line);
    }
    while(true);
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
}
```

Zurück zum Netzwerkbeispiel:

```
public class TCPClient
{
  public static void main(String[] args)
  {
    final int PORT = 5000;
    final String HOST = "localhost";
    try (Socket connectionToServer = new Socket(HOST, PORT);
         OutputStream os = connectionToServer.getOutputStream();)
    {
      os.write("Suppe mit dem Löffel löffeln\n".getBytes()); //Charset.defaultCharset()
      os.write("Suppe mit dem Löffel löffeln\n".getBytes(StandardCharsets.ISO_8859_1));
      os.write("Suppe mit dem Löffel löffeln\n".getBytes(StandardCharsets.UTF_16));
      os.write("Suppe mit dem Löffel löffeln\n".getBytes(StandardCharsets.UTF_8));
      os.write("Suppe mit dem Löffel löffeln\n".getBytes(StandardCharsets.US_ASCII));
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }
}
```

```
public class TCPServer {
  public static void main(String[] args) {
    final int PORT = 5000;
    try(ServerSocket ss = new ServerSocket(PORT);
        Socket connection = ss.accept();
        InputStream is = connection.getInputStream();
        BufferedReader br = new BufferedReader(new InputStreamReader(is));)
    {
      System.out.println(Charset.defaultCharset()    + ": " + br.readLine());
      System.out.println(StandardCharsets.ISO_8859_1 + ": " + br.readLine());
      System.out.println(StandardCharsets.UTF_16 + ": " + br.readLine());
      System.out.println(StandardCharsets.UTF_8 + ": " + br.readLine());
      System.out.println(StandardCharsets.US_ASCII + ": " + br.readLine());
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
  }
}
```



Wie sind Streams aufgebaut?

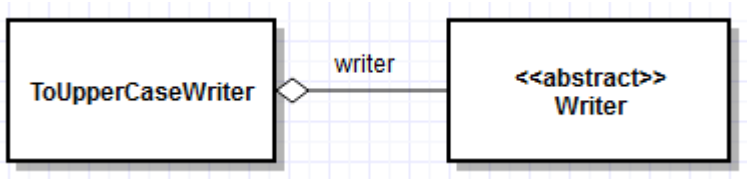
Decorator Pattern

Streams erweitern oft ihre Funktionalität,
indem Sie Code von bestehenden Streams verwenden.

Nehmen wir an, wir wollen einen Character Stream schreiben,
der alle Buchstaben in Großbuchstaben verwandelt.

```
public class ToUpperCaseWriter {
  Writer writer;

  public ToUpperCaseWriter(Writer writer) {
    this.writer = writer;
  }
  public void write(char c) throws IOException {
    writer.write(Character.toUpperCase(c));
  }
  public void write(char[] cbuf, int off, int len) throws IOException {
    for (int i = off; i < off + len; i++)
      write(cbuf[i]);
  }
  public void write(String str, int off, int len) throws IOException {
    for (int i = off; i < off + len; i++)
      write(str.charAt(i));
  }

  … //alle (relevanten) Methoden von Writer überschreiben

  public void flush() throws IOException {
    writer.flush();
  }
  public void close() throws IOException {
    writer.close();
  }
}
```

Wiederverwendung von Writer durch Komposition/Aggregation



```
public static void main(String[] args) throws Exception
{
  FileWriter fw = new FileWriter("atouppercasefile.txt");
  ToUpperCaseWriter tw = new ToUpperCaseWriter(fw);
  tw.write("Suppe mit dem Löffel löffeln");
  tw.close();
}
```

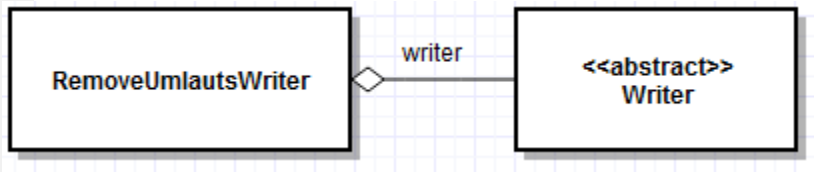
Wie verwenden wir den Writer?

Nehmen wir an, wir wollen einen weiteren Character Stream schreiben,
der alle Umlaute ersetzt.

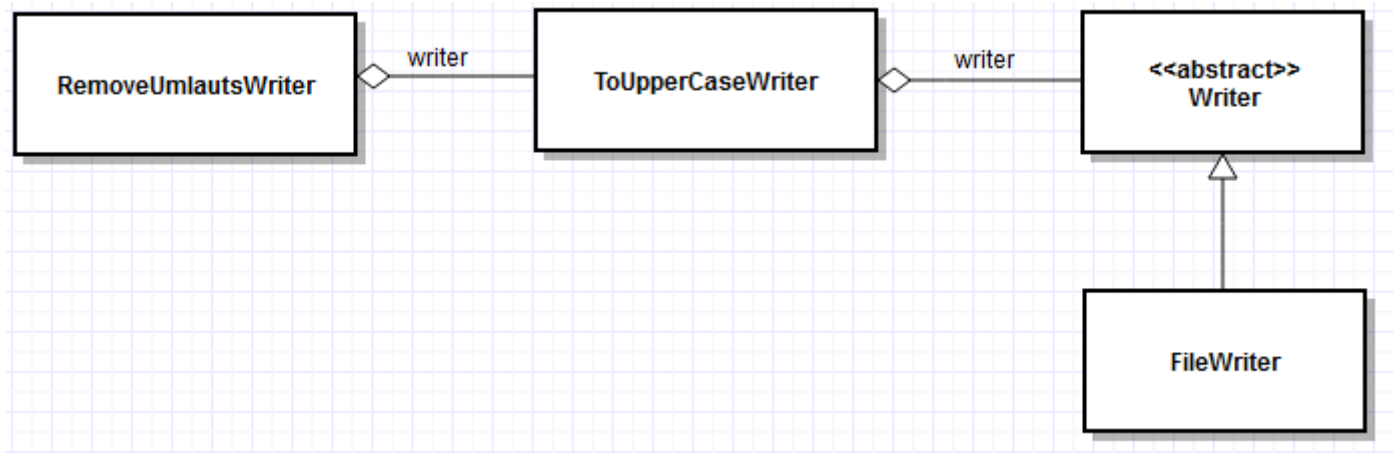
ä -> ae, ö -> oe, ü -> ue
Ä -> Ae, Ö -> Oe, Ü -> Ue

```
public class RemoveUmlautsWriter {
  Writer writer;

  public RemoveUmlautsWriter(Writer writer) {
    this.writer = writer;
  }

  public void write(char c) throws IOException {
    if (c == 'ä') {
      writer.write('a');
      writer.write('e');
    }
    else if (c == 'ö') {
      writer.write('o');
      writer.write('e');
    }
    else if (c == 'ü') {
      writer.write('u');
      writer.write('e');
    }
    … //Großbuchstaben
    else writer.write(c);
  }

  … //alle (relevanten) Methoden von Writer überschreiben

}
```

Wiederverwendung von Writer durch Komposition/Aggregation



Wie bekommen wir es hin, dass wir beide Writer verwenden können,
um zunächst die Umlaute zu ersetzen
und dann alles in Großbuchstaben umzuwandeln?



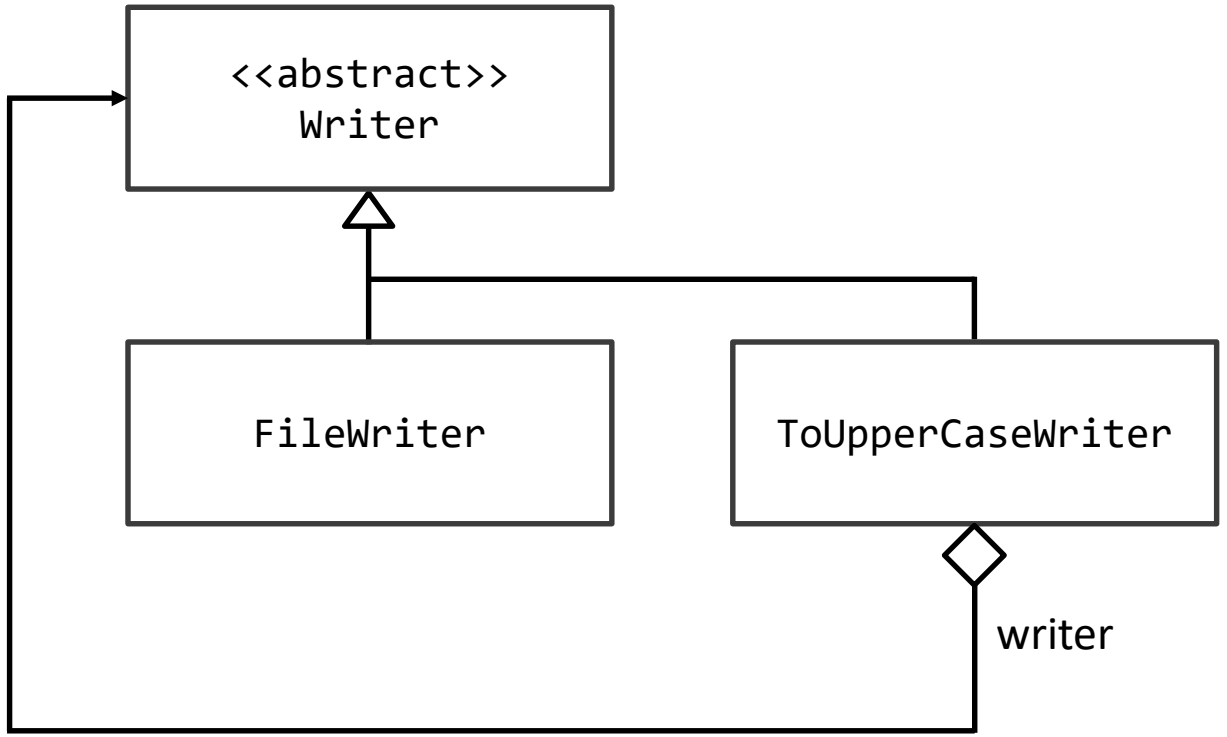
Wir wollen unsere Writer also wie folgt benutzen:

```
RemoveUmlautsWriter rw =
new RemoveUmlautsWriter(new ToUpperCaseWriter(new FileWriter("rwtwfw.txt")));
```

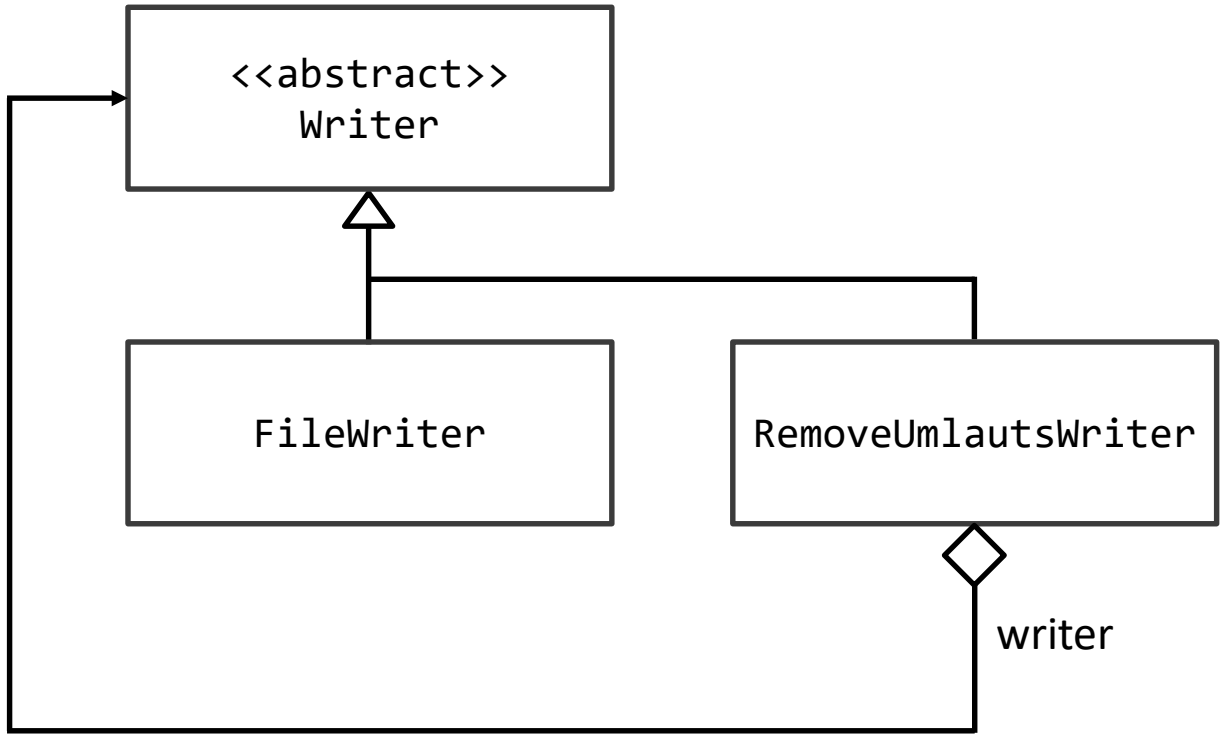
Wenn unsere Writer echte Writer wären, d.h. von Writer erbten,
könnte RemoveUmlautsWriter einfach ToUpperCaseWriter verwenden.

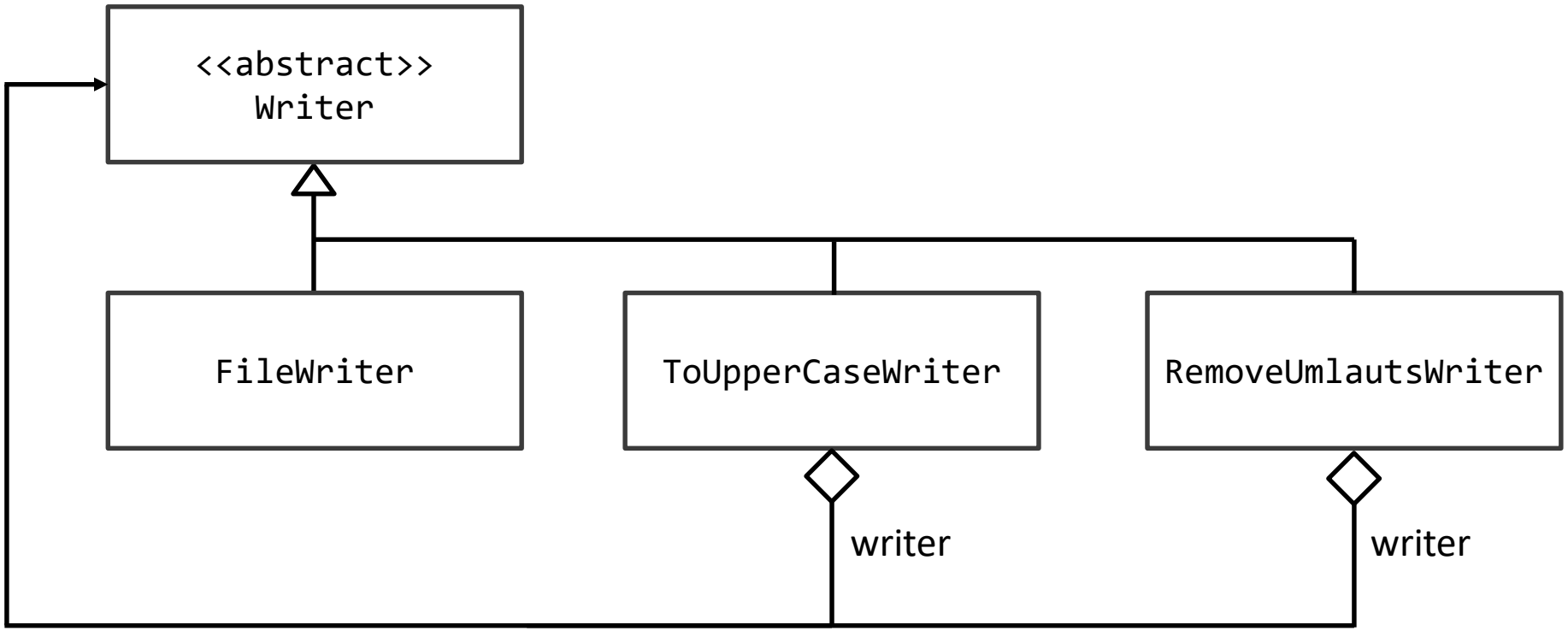
Ist RemoveUmlautsWriter **ein** Writer? (Vererbungsfrage)

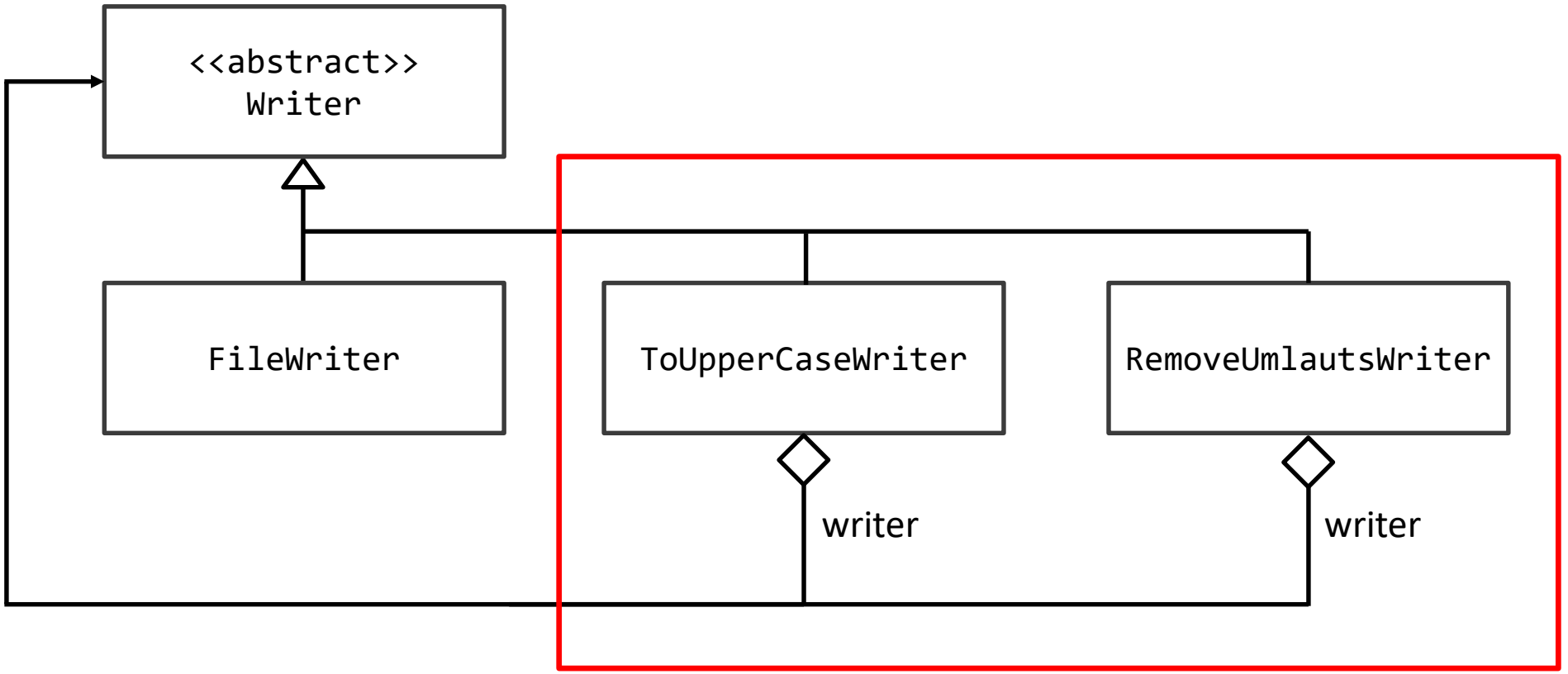
```
public class ToUpperCaseWriter extends Writer {
  Writer os;

  public ToUpperCaseWriter(Writer os) {
    this.os = os;
  }
  public void write(char c) throws IOException {
    os.write(Character.toUpperCase(c));
  }
  public void write(char[] cbuf, int off, int len
    for (int i = off; i < off + len; i++)
      write(cbuf[i]);
  }
  public void write(String str, int off, int len)
    for (int i = off; i < off + len; i++)
      write(str.charAt(i));
  }

  … //alle (relevanten) Methoden von Writer übers

  public void flush() throws IOException {
    os.flush();
  }
  public void close() throws IOException {
    os.close();
  }
}
```



```
public class RemoveUmlautsWriter extends Writer {
  Writer writer;

  public RemoveUmlautsWriter(Writer writer) {
    this.writer = writer;
  }

  public void write(char c) throws IOException {
    if (c == 'ä') {
      writer.write('a');
      writer.write('e');
    }
    else if (c == 'ö') {
      writer.write('o');
      writer.write('e');
    }
    else if (c == 'ü') {
      writer.write('u');
      writer.write('e');
    }
    … //Großbuchstaben
    else writer.write(c);
  }

  … //alle (relevanten) Methoden von Writer übersc

}
```

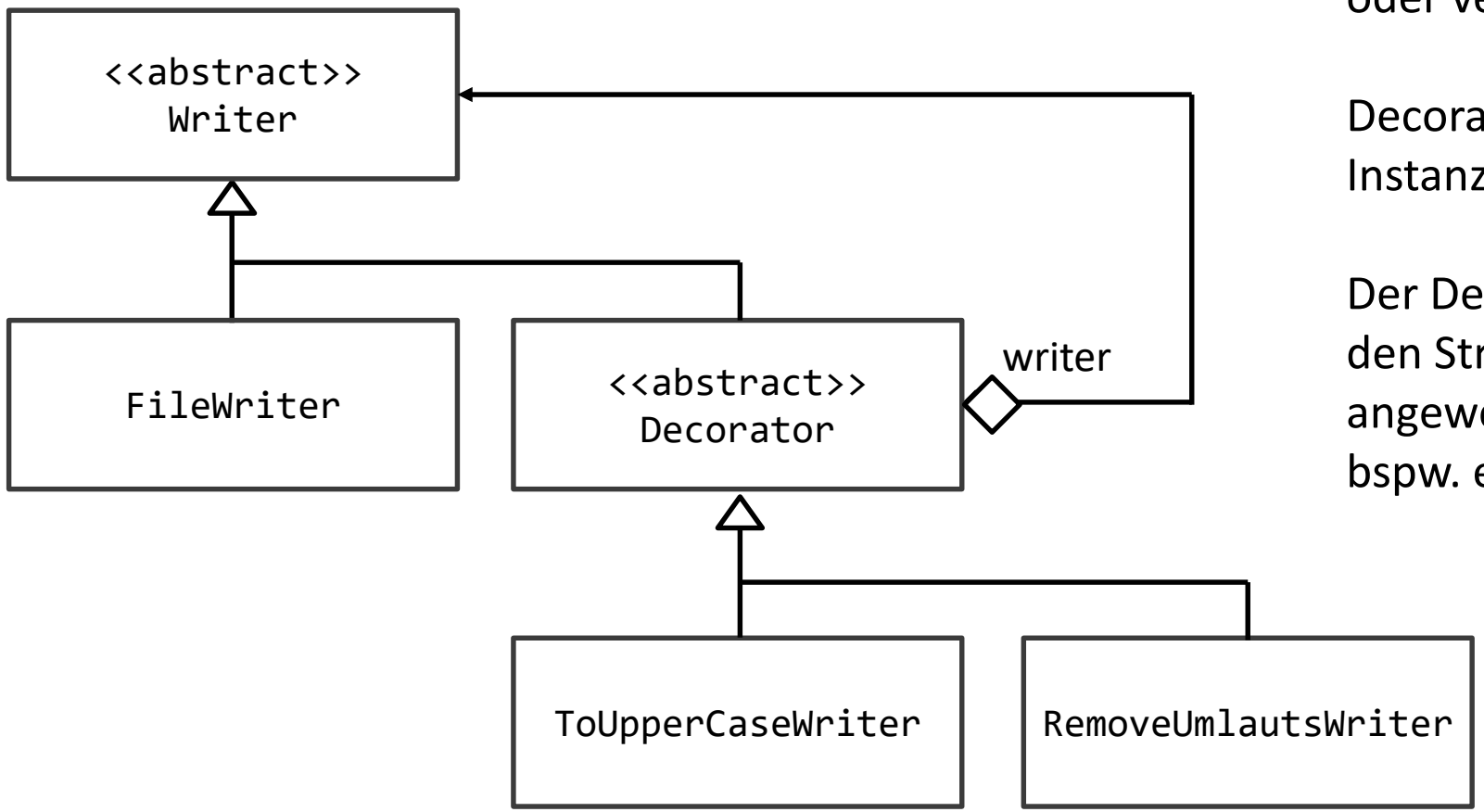






Um nicht für jeden konkreten Dekorator eine Kompositionsbeziehung modellieren zu müssen, fügt man eine abstrakte Klasse ein.

Decorator Pattern



Ein Decorator fügt der dekorierten Instanz weitere Funktionalitäten hinzu.

Ein Decorator besitzt mindestens dieselbe öffentliche Schnittstelle wie die dekorierte Instanz, leitet Aufrufe an diese Instanz weiter oder verarbeitet die Aufrufe modifiziert.

Decorator können auch bereits dekorierte Instanzen dekorieren.

Der Decorator-Ansatz wird auch zum Teil bei den Streams der Java-Klassenbibliothek angewendet: Der BufferedReader dekoriert bspw. einen Reader.

Decorator Pattern

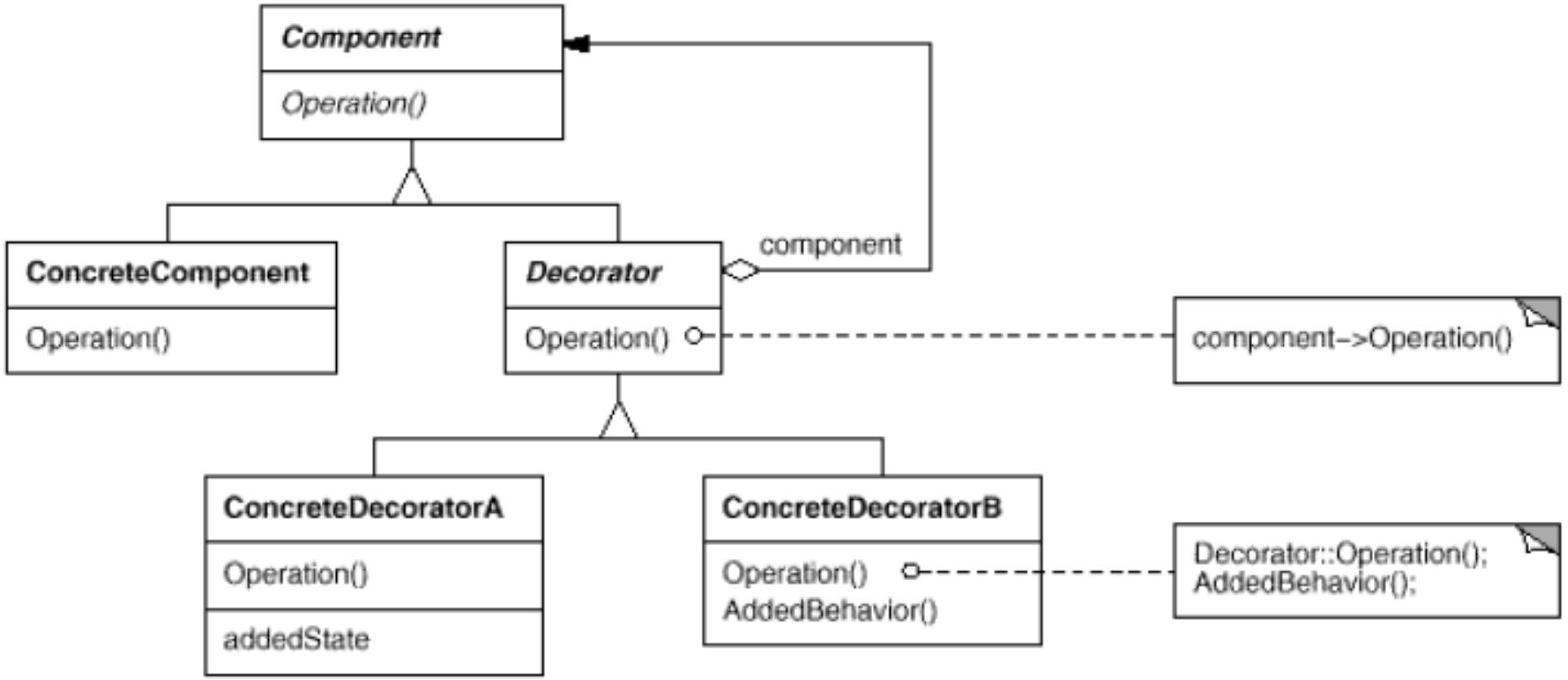


Bild von: Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software, S. 177, ISBN: 978-0201633610

Unit Tests II

Nehmen wir einen unserer Decorator-Streams
als weiteres Beispiel!

Wir wollen unsere Writer mit JUnit Tests testen.

Wenn wir bspw. ein a in den Stream schreiben,
müssen wir ein A in der Datensenke finden.

Wie gehen wir vor?

```
public class ToUpperCaseWriterTest
{
  @Test
  public void writeCharTest()
  {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    OutputStreamWriter osw = new OutputStreamWriter(baos);
    ToUpperCaseWriter writer = new ToUpperCaseWriter(osw);
    try
    {
      writer.write('a');
      writer.flush();
      String uppercase = baos.toString();
      assertEquals("A", uppercase);
      writer.close();
    }
    catch (IOException e)
    {
      fail("IOException" + e.getMessage());
    }
  }
}
```

assertEquals testet anhand der equals-Methode, ob beide Objekte gleich sind. Ansonsten schlägt der Test fehl.

Hier findet der eigentliche Test statt:
Der Test läuft durch,
wenn der Inhalt von uppercase “A” entspricht.
Ansonsten schlägt er fehl.

fail lässt den Test fehlschlagen

Das war aber nur ein Testfall.

```
public class ToUpperCaseWriterTest
{
  @Test
  public void writeCharTestWithNonChar()
  {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    OutputStreamWriter osw = new OutputStreamWriter(baos);
    ToUpperCaseWriter writer = new ToUpperCaseWriter(osw);
    try
    {
      writer.write('1');
      writer.flush();
      String uppercase = baos.toString();
      assertEquals("1", uppercase);
      writer.close();
    }
    catch (IOException e)
    {
      fail("IOException" + e.getMessage());
    }
  }
}
```

Hier findet der eigentliche Test statt:
Der Test gelingt, wenn der Inhalt (1) unverändert bleibt.
Ansonsten schlägt er fehl.

Die beiden Tests haben sehr viel **doppelten Code.**

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
OutputStreamWriter osw = new OutputStreamWriter(baos);
ToUpperCaseWriter writer = new ToUpperCaseWriter(osw);
try
{
  writer.write('1');
  writer.flush();
  String uppercase = baos.toString();
  assertEquals("1", uppercase);
  writer.close();
}
catch (IOException e)
{
  fail("IOException" + e.getMessage());
}
```

Die Ausgangssituation der Tests ist bspw. identisch.

Da die Ausgangssituation für verschiedene Tests häufig identisch ist, wurde die Annotation @BeforeEach eingeführt.

```
public class ToUpperCaseWriterTest
{
  ToUpperCaseWriter writer;
  ByteArrayOutputStream baos;

  @BeforeEach
  public void prepareTest()
  {
    baos = new ByteArrayOutputStream();
    OutputStreamWriter osw =
     new OutputStreamWriter(baos);
    writer = new ToUpperCaseWriter(osw);
  }

  …
}
```

Eine mit @BeforeEach annotierte Methode wird vor jedem Test ausgeführt.

```
@Test
public void writeCharTest() {
  try {
    writer.write('a');
    writer.flush();
    String uppercase = baos.toString();
    assertEquals("A", uppercase);
  }
  catch (IOException e) {
    fail("IOException" + e.getMessage());
  }
}

@Test
public void writeCharTestWithNonChar() {
  try {
    writer.write('1');
    writer.flush();
    String uppercase = baos.toString();
    assertEquals("1", uppercase);
  }
  catch (IOException e) {
```

Mit @AfterEach können analog nach einem Test Aufräumarbeiten durchgeführt werden.

```
public class ToUpperCaseWriterTest
{
  ToUpperCaseWriter writer;
  ByteArrayOutputStream baos;

  @AfterEach
  public void cleanUp()
  {
    try
    {
      writer.close();
    }
    catch (IOException e)
    {
      fail("IOException" + e.getMessage());
    }
  }
  …
}
```

Eine mit @AfterEach annotierte Methode wird nach jedem Test ausgeführt.

```
@Test
public void writeCharTest() {
  try {
    writer.write('a');
    writer.flush();
    String uppercase = baos.toString();
    assertEquals("A", uppercase);
  }
  catch (IOException e) {
    fail("IOException" + e.getMessage());
  }
}

@Test
public void writeCharTestWithNonChar() {
  try {
    writer.write('1');
    writer.flush();
    String uppercase = baos.toString();
    assertEquals("1", uppercase);
  }
  catch (IOException e) {
```

Test-Driven Development

Idea: Write test code before production code.

Three Laws of TDD (summarized)

First Law: Write no production code, except to pass a failing test.

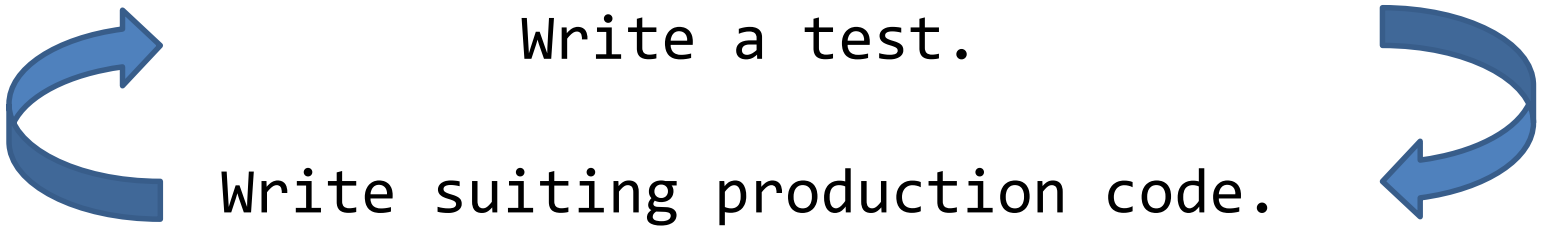
Second Law: Write only enough of a test to demonstrate a failure.

Third Law: Write only enough production code to pass a test.

source: http://agileinaflash.blogspot.de/2009/03/unclebobs-three-rules-of-tdd.html

source: Robert C. Martin: Clean Code, Prentice Hall, 2008, ISBN: 978-0132350884

Development happens in short cycles.



Leads to dozens of tests a day.

Leads to hundreds of tests a month.

Leads to thousands of tests a year.

Leads to dozens of tests a day.

Leads to hundreds of tests a month.

Leads to thousands of tests a year.

Leads to dozens of tests a day.

Leads to hundreds of tests a month.

Leads to thousands of tests a year.

Tests will cover virtually all of the production code!

Test code is as important as production code.

Test code must be kept as clean as production code.

Without tests you are afraid to touch working code. Each change is a potential bug.

With tests you don’t fear to change (improve) your code.

The higher the test coverage, the less fear you need to have.

- Test code allows keeping production code flexible, maintainable, and reusable.
- Tests enable change!