

Einführung in die Informatik I
Ein Beispiel

Prof. Bernd Brügge, Ph.D
Technische Universität München

Wintersemester 2000/2001

5. und 6. Februar 2001

Überblick über diesen Vorlesungsblock

- ❖ **Benutzung von abstrakten Methoden mit einem etwas komplizierteren Modellierungsbeispiel: Elektronische Authentifizierung**
- ❖ **Themen:**
 - Typkonvertierung in Java
 - Kodierung von Zeichen (Unicode)
 - Unterschied Gleichheit und Identität
 - am Beispiel von Zeichenketten
 - Zwei weitere Klassen/Interfaces:
 - **Enumeration, StringTokenizer**
 - Aufbau der Java-Klassenbibliothek
 - Packages
 - Polymorphismus:
 - Überschreiben von Methoden
 - Ein weiteres Entwurfsmuster: Brückenmuster

Ziele

- ❖ Sie können aktiv mit abstrakten Methoden umgehen.
- ❖ Sie haben ein erstes Verständnis der Klassenbibliothek in Java und können mit ihr umgehen (Beispiel: Klassen und Methoden suchen/benutzen).
- ❖ Sie verstehen den Unterschied zwischen Gleichheit und Identität
- ❖ Sie verstehen das Überschreiben von Methoden.
- ❖ Sie können besser modellieren, insbesondere haben Sie ein besseres Verständnis, wie Vererbung in der Modellierung benutzt werden kann.

Problembeschreibung

- ❖ Das Studentenverwaltungssystem soll nur von **authorisierten Benutzern** benutzt werden können.
 - Jeder Benutzer hat ein Passwort. Das Passwort muss vom Benutzern getippt werden, wenn sie das System benutzen wollen.
- ❖ Das **Passwort** soll nicht von anderen lesbar sein. Es muss deshalb **verschlüsselt** werden.
- ❖ Das System soll **verschiedene Chiffrier-Verfahren** anbieten. Bei der ersten Lieferung sollen bereits folgende Verfahren mitgeliefert werden:
 - **Caesar-Verfahren**
 - **Transpositionsverfahren**
- ❖ Das System soll **erweiterbar** sein: Wenn bessere Chiffrierverfahren auf den Markt kommen, dann sollen diese Verfahren in das Campus-System eingebunden werden können, ohne dass das gesamte System neu kompiliert werden muss.

Analyse der Anforderungen

❖ **Akteure:**

- Autorisierter Benutzer:
 - Administrator (Immatrikulationsamt, Prüfungsausschuss)
 - Normaler Benutzer (Student)

❖ **Funktionale Anforderungen:**

- Einloggen in das Campus-System mit Benutzernamen, Passwort
- Chiffrieren und Dechiffrieren von Passwörtern mit Caesar und Transposition

❖ **Nichtfunktionale Anforderungen:**

- **Erweiterbarkeit:** Unterstützung von verschiedenen Verschlüsselungsmethoden, insbesondere Caesar und Transposition, aber auch solche, die bei der Problemstellung noch nicht bekannt sind.

Kryptographische Verfahren

- ❖ Kryptographische Verfahren benutzen im Allgemeinen eine Chiffre mit zwei Methoden.
 - **Chiffrierung/Verschlüsselung:** Die Umwandlung von Daten in irgendeine unlesbare Form. Damit soll bezweckt werden, dass niemand die Information erlangen kann, auch wenn er die verschlüsselten Daten sieht.
 - **Dechiffrierung/Entschlüsselung:** Die Umkehrung der Verschlüsselung. Hierbei werden die verschlüsselten Daten zurück in eine auswertbare Form umgewandelt.
- ❖ Es gibt viele Algorithmen zur Implementierung dieser Methoden. Wir schauen uns nur zwei relativ unkomplizierte Algorithmen an:
 - Substitutions-Chiffre
 - Transpositions-Chiffre

Anwendungsbereiche der Kryptographie

- ❖ Kryptographie kann auch für andere Bereiche benutzt werden:
 - Eine **elektronische Unterschrift** verbindet ein Dokument mit dem Eigentümer eines bestimmten Schlüssels.
 - Ein **digitaler Zeitstempel** bindet ein Dokument bei seiner Erstellung an eine bestimmte Zeit (kann man benutzen, um einen Pay-TV Kanal einzurichten).
 - **Elektronisches Geld** erlaubt uns, im Internet zu kaufen und zu bezahlen.
- ❖ Die Kryptographie spielt eine fundamentale Rolle im Zugriff auf Informationssysteme und bei der Übertragung von Nachrichten.
 - **Buch:** F.L. Bauer, Entzifferte Geheimnisse, Springer Verlag.
 - **Vorlesungen im Hauptstudium:** A. Gerold, Kryptologie I und Kryptologie II (<http://www.in.tum.de/~gerold/krypto-ss2001.html>)

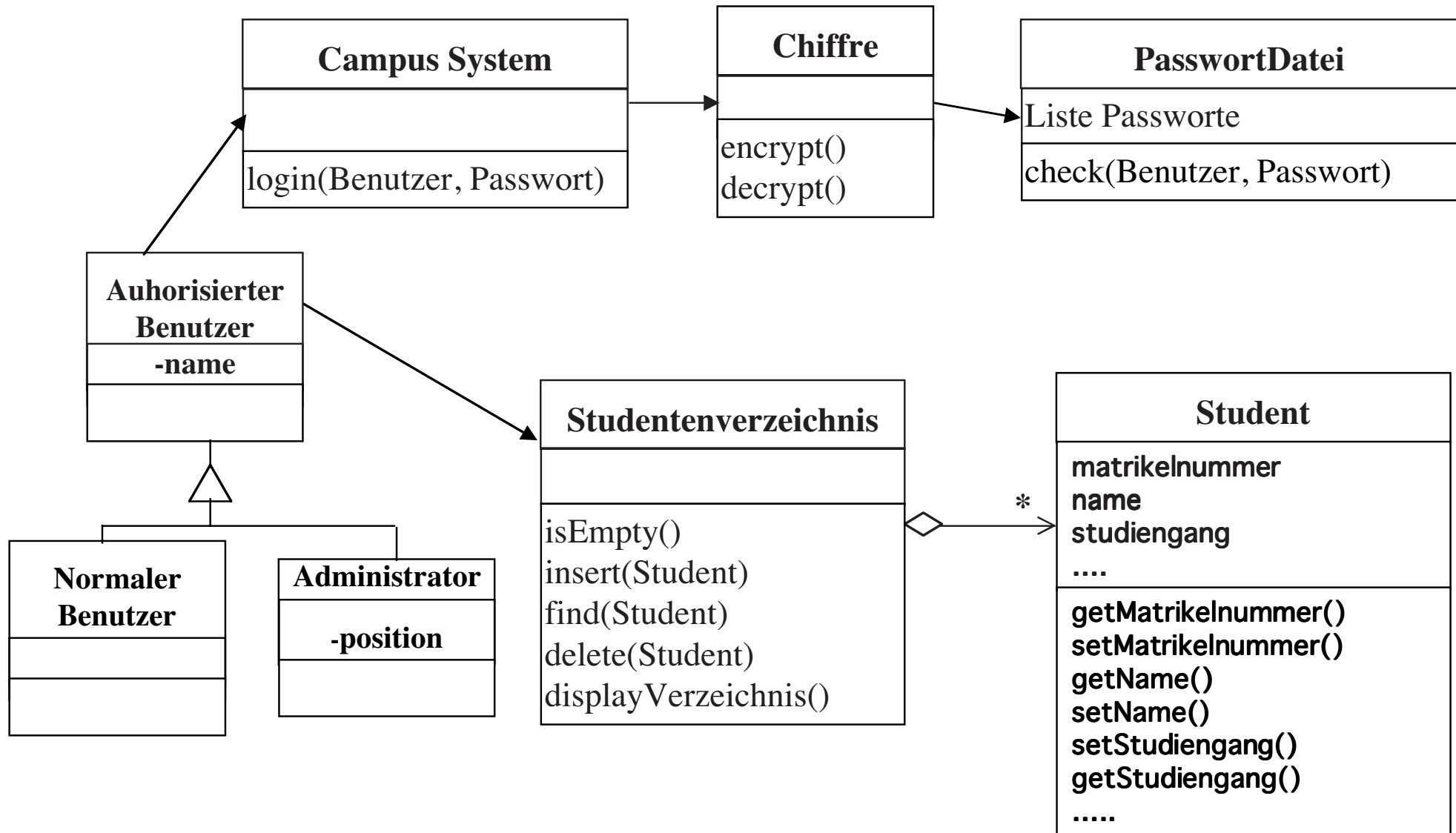
Substitutions-Chiffre

- ❖ Eine Substitutions-Chiffre ersetzt jeden Buchstaben in einem Textblock durch einen Buchstaben aus dem **Substitutionsalphabet**. Beispiele:
- ❖ **Caesar-Chiffre** (Caesar benutzte sie im Gallischen Krieg)
 - Normales Alphabet: **abcdefghijklmnopqrstuvwxy**z
 - Substitutionsalphabet: **defghijklmnopqrstuvwxyzabc**
 - Verschlüsselungsbeispiel: **hello** → **khoor**
- ❖ **Substitutions-Chiffre mit Schlüsselwort xylophn** :
 - Normales Alphabet: **abcdefghijklmnopqrstuvwxy**z
 - Substitutionsalphabet: **xylophnabcdefghijklmnopgi**jk~~lm~~qrstuvwz
 - Verschlüsselungsbeispiel: **hello** → **apeei**

Transpositionsverfahren

- ❖ Eine Transpositions-Chiffre transformiert die Buchstaben in einem Textblock nach einem bestimmten Algorithmus durch andere Buchstaben.
- ❖ **Rotations-Chiffre:** Die Buchstaben in jedem Wort werden rotiert (die Reihenfolge der Buchstaben wird umgekehrt).
 - Verschlüsselungsbeispiel: **hello world** → **olleh dlrow**
- ❖ **3-Buchstaben-Shift-Chiffre:** Die Buchstaben in jedem Wort werden um 3 Buchstaben modulo der Wortlänge verschoben:
 - Verschlüsselungsbeispiel: **hello world** → **lohel ldwor**

Analyse des Problems: Klassendiagramm

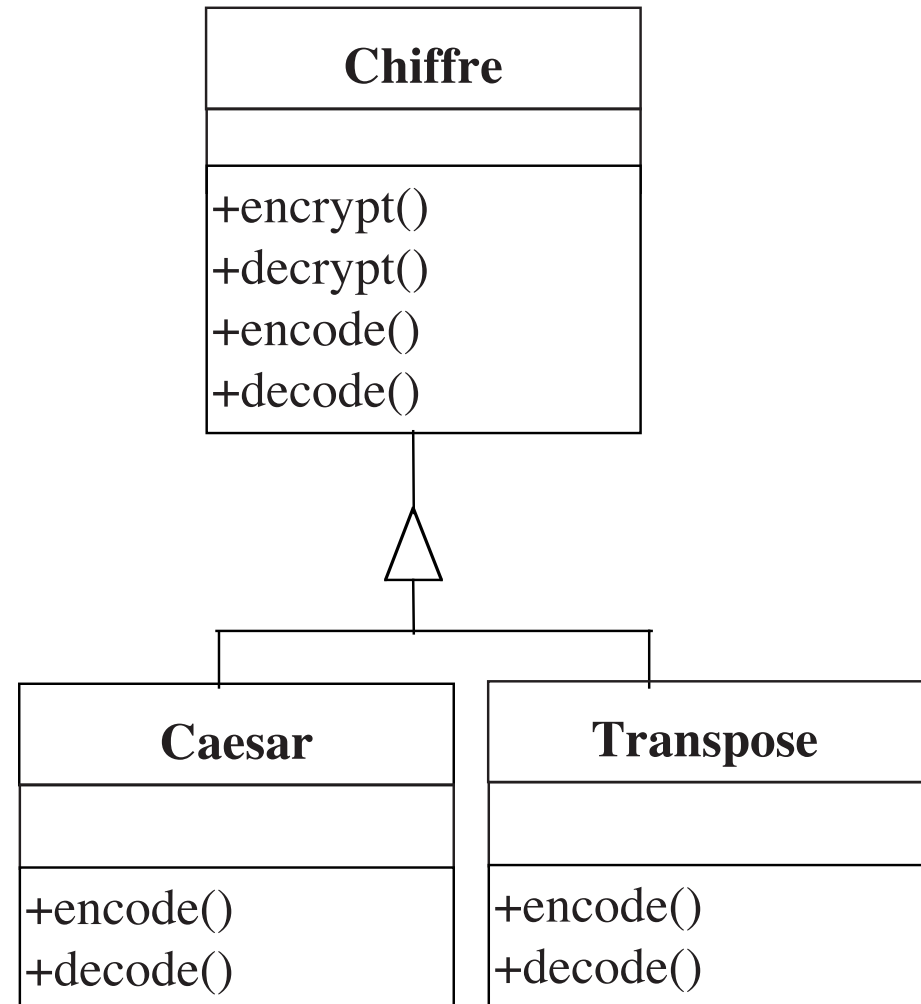


Objekt-Entwurf

- ❖ Wir definieren **Chiffre** als Superklasse, und die einzelnen Chiffrierverfahren als Unterklassen von **Chiffre**.

Chiffre bietet 4 Methoden an:

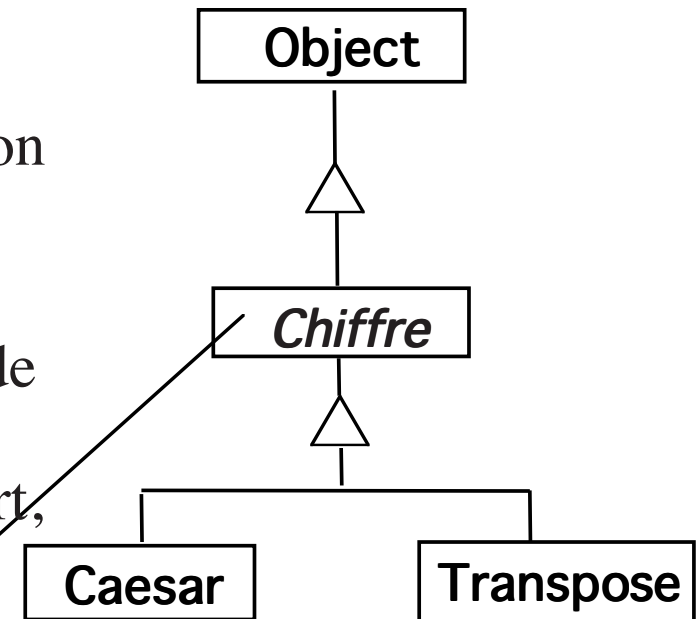
- **encrypt()** verschlüsselt einen Text bestehend aus Wörtern.
- **decrypt()** entschlüsselt einen Text bestehend aus Wörtern.
- **encode()** wendet einen spezifischen Algorithmus zur Verschlüsselung eines Wortes an.
- **decode()** wendet einen spezifischen Algorithmus zur Entschlüsselung eines Wortes an.



Implementation in Java

- ❖ Die encrypt- und decrypt-Methoden sind dieselben für jede Unterklasse und können deshalb in der Superklasse **Chiffre** *implementiert* werden.
 - **Chiffre** definieren wir als Unterklasse von **Object**, weil wir einige Methoden von **Object** benötigen.
- ❖ Die encode- und decode-Methoden sind für jede Unterklasse spezifisch. Sie werden deshalb als *abstrakte Methoden* in der Superklasse definiert, und dann in den Unterklassen *implementiert*.

Die Chiffre-Klasse ist abstrakt, weil einige ihrer Methoden noch nicht implementiert sind.



Zusätzliche Java-Konzepte

- ❖ Java's Zeichensatz: Unicode
- ❖ Java-Typkonvertierungen
- ❖ Vergleich von Zeichenketten
 - Identität und Gleichheit
- ❖ Java-Klasse: **StringTokenizer**

Zeichen

- ❖ Der Typ **char** ist ein primitiver Datentyp in Java.
 - In Java werden Werte vom Typ **char** durch eine 16-bit Ganzzahl ohne Vorzeichen repräsentiert.
 - Mit 16 bit kann man $2^{16} = 65536$ Zeichen repräsentieren.
- ❖ Java benutzt den internationalen **Unicode**-Zeichensatz zur Kodierung von Zeichen:
 - Der Unicode-Zeichensatz wurde entwickelt, um Zeichenketten in allen internationalen Sprachen darstellen zu können, nicht nur in Englisch.
 - Detaillierte Informationen unter: **<http://www.unicode.org>**
 - Um rückwärtskompatibel zu sein, sind die ersten 128 Unicode-Zeichen mit den Zeichen des 7-bit Zeichensatzes ASCII (American Standard Code for Information Interchange) identisch.

Die Zeichen des ASCII-Zeichensatzes und ihre Ganzzahlwerte

Zeichen	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
Code	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Zeichen	0	1	2	3	4	5	6	7	8	9						
Code	48	49	50	51	52	53	54	55	56	57						
Zeichen	:	;	<	=	>	?	@									
Code	58	59	60	61	62	63	64									
Zeichen	A	B	C	D	E	F	G	H	I	J	K	L	M			
Code	65	66	67	68	69	70	71	72	73	74	75	76	77			
Zeichen	N	O	P	Q	R	S	T	U	V	W	X	Y	Z			
Code	78	79	80	81	82	83	84	85	86	87	88	89	90			
Zeichen	[\]	^	_	`										
Code	91	92	93	94	95	96										
Zeichen	a	b	c	d	e	f	g	h	i	j	k	l	m			
Code	97	98	99	100	101	102	103	104	105	106	107	108	109			
Zeichen	n	o	p	q	r	s	t	u	v	w	x	y	z			
Code	110	111	112	113	114	115	116	117	118	119	120	121	122			
Zeichen	{		}	~												
Code	123	124	125	126												

Zeichen → Ganzzahl-Konvertierung

- ❖ Werte vom Typ **char** können sowohl durch Zeichen als auch durch Zahlen aus dem Unicode repräsentiert werden.

```
char ch = 'a';  
System.out.println(ch);           // Druckt 'a'  
System.out.println((int)'a') ;   // Druckt 97
```

Konvertierungsoperator

- ❖ Der Konvertierungsoperator (*cast operator*) konvertiert einen Datentyp ('a') in einen anderen (97).

Typkonvertierung

- ❖ **Definition:** Eine **Typkonvertierung** (type cast) konvertiert einen Typ in einen anderen.

Die Konvertierung wird mittels eines Typ-Bezeichners gemacht, der geklammert vor dem zu konvertierenden Ausdruck steht.

- ❖ Beispiele:

```
- char c;           // Variable c vom Type char
- int k;           // Variable k vom Typ int
- c = 'a';        // Zuweisung des Wertes 'a' an Variable c
- k = (int) c;    // Die Variable k bekommt den int-Wert von a
                  // Der int-Wert ist 97, der sogenannte
                  // Unicode-Wert (oder ASCII-Wert) von 'a'
```

Andere Typkonvertierungen in Java

- ❖ Typkonvertierung in Java folgt vielen Regeln und Konventionen. In einigen Fällen kann der Programmierer sogar *implizite Typkonvertierungen* machen. Beispiele:

char → *int*:

```
char ch;  
int k;  
k = ch; // konvertiert einen Buchstaben  
        // in eine 32-Bit-Zahl
```

int → *double*:

```
int i;  
double d;  
d = i; // konvertiert eine 32-Bit-Zahl  
        // in eine 64-Bit-Zahl
```

- ❖ In anderen Fällen muss der Programmierer *explizite Typkonvertierungen* machen. Beispiele

double → *int*:

```
int i;  
double d;  
i = d; // geht nicht implizit!  
        // Eine Zahl vom Typ double passt  
        // nicht in eine Zahl vom Typ int.  
        // Explizite Typkonvertierung:  
i = (int) d;
```

int → *String*: mit *valueOf()*

```
String s;  
int i;  
i = 45;  
s = String.valueOf(i);
```

Die primitiven Java-Typen und ihre Wortlänge in Bits

Datentyp	Schlüsselwort	Wortlänge (Bits)
Bool	boolean	-
Byte	byte	8
Zeichen	char	16
Ganzzahl	short	16
Ganzzahl	int	32
Ganzzahl	long	64
Gleitkommazahl	float	32
Gleitkommazahl	double	64

Regeln für Typkonvertierungen

- ❖ Java erlaubt implizite Typkonvertierungen für primitive Typen von einem *engeren* (weniger bits) zu einem *weiteren Typ* (mehr bits).
- ❖ Ein Konvertierungsoperator (cast operator) muss benutzt werden, wenn man einen weiteren in einen engeren Typ konvertieren möchte.
 - Der Operator kann mit jedem beliebigen primitiven Typ benutzt werden. Er bezieht sich auf den Bezeichner oder Ausdruck, der dem Operator unmittelbar folgt:

```
int n = 4;  
char m = '5';  
char ch = (char)(m + n); // Konvertiere die Summe von m und n in ein Zeichen
```

Identität vs Gleichheit am Beispiel von Zeichenketten

- ❖ Für beliebige Java-Objekte o1 und o2 ist die Evaluierung des booleschen Ausdrucks o1 == o2 wahr, wenn o1 und o2 *identisch* sind, d.h. die Referenzvariablen haben denselben Wert (zeigen auf dasselbe Objekt!)
- ❖ Wenn man 2 Zeichenketten vergleichen will, darf man nicht einfach den "=="-Operator verwenden:

```
String s1 = "hello"; String s2 = "Hello";  
if (s1 == s2)..... // vergleicht die Referenzwerte von s1 und s2,  
                    // nicht die Zeichenketten, auf die s1 und s2 zeigen
```

Definition: 2 Zeichenketten sind gleich, wenn sie die gleichen Zeichen in derselben Reihenfolge enthalten.

- ❖ Zum Vergleich von Zeichenketten bietet Java folgende Methoden an:

```
public boolean equals(Object anObject);    // Überschreibt Object.equals()  
public boolean equalsIgnoreCase(String anotherString)  
public int compareTo(String anotherString)
```

Identität vs. Gleichheit von Zeichenketten

Mit diesen Deklarationen...

```
String s1 = new String("hello");  
String s2 = new String("hello");  
String s3 = new String("Hello");  
String s4 = s1;  
String s5 = "hello";  
String s6 = "hello";
```

s5 und s6
bezeichnen
dieselbe Konstante

bekommen wir folgende Resultate...

Gleichheit:

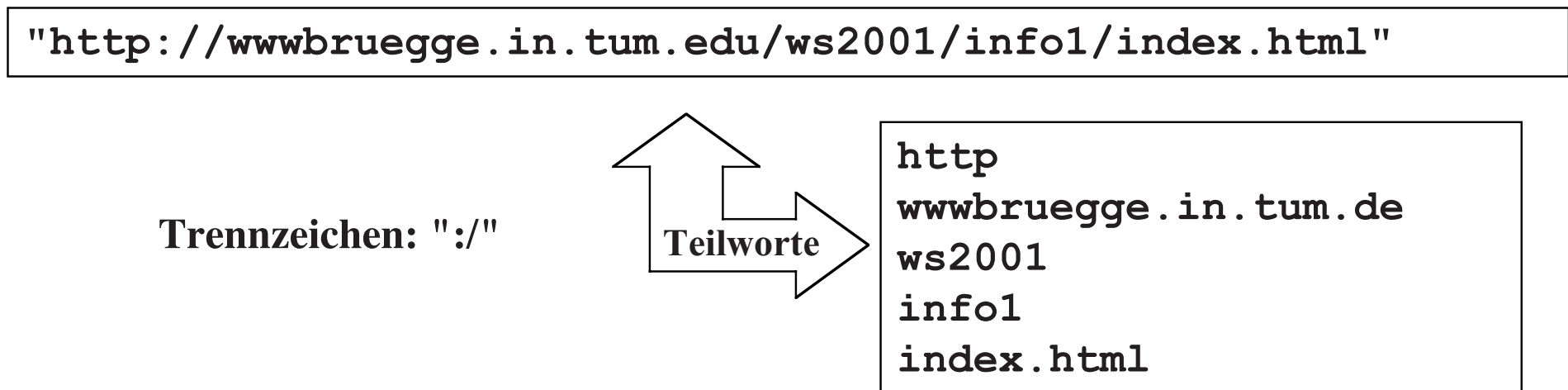
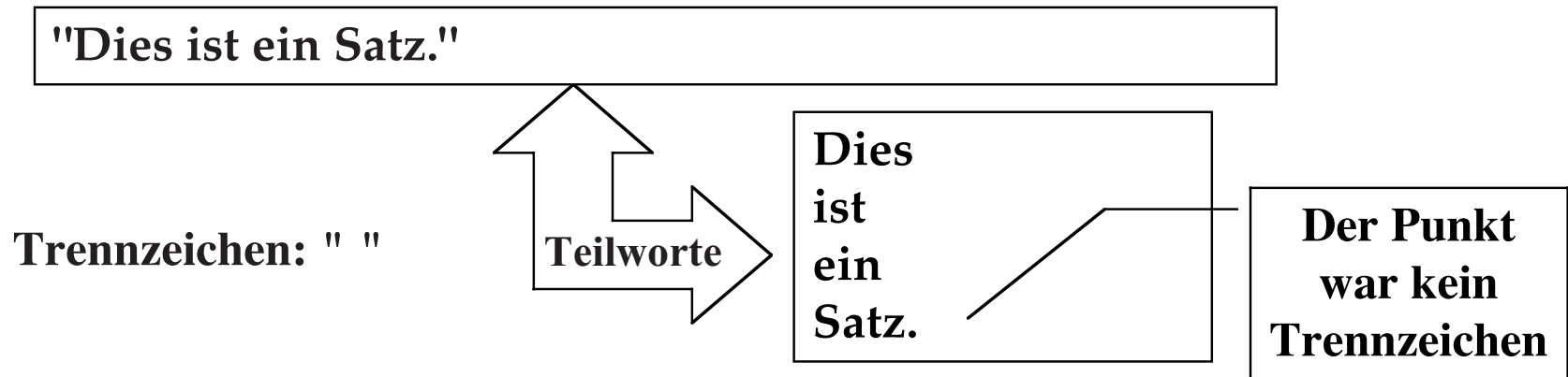
```
s1.equals(s2)           ==> true  
s1.equals(s3)          ==> false  
s1.equalsIgnoreCase(s3) ==> true  
s1.equals(s4)          ==> true  
s1.equals(s5)          ==> true  
s1.compareTo(s6)       ==> 0
```

Identität:

```
s1 == s2           ==> false  
s1 == s3           ==> false  
s1 == s4           ==> true  
s1 == s5           ==> false  
s5 == s6           ==> true
```

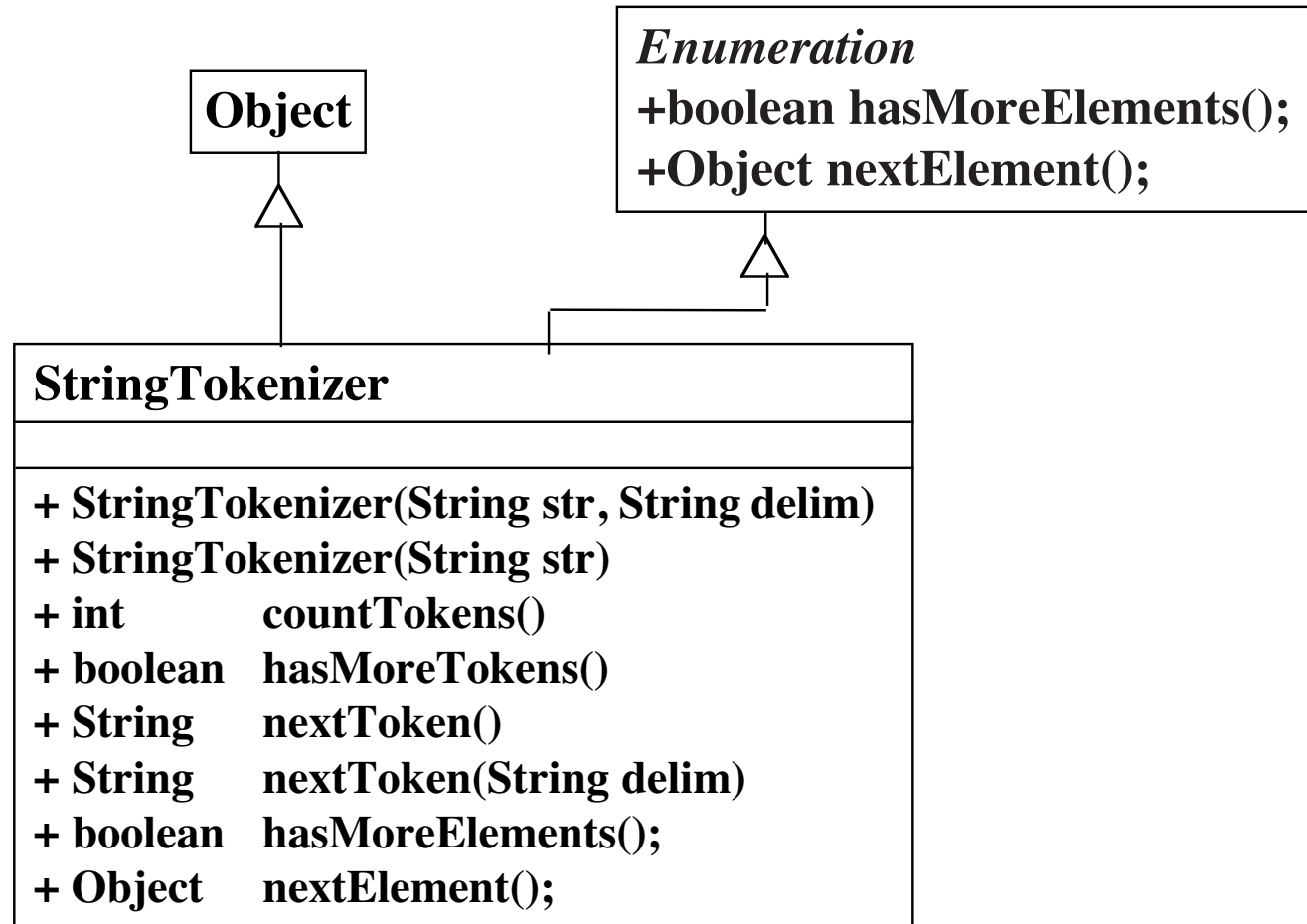
Für die Verschlüsselung müssen wir die Zeichenkette in Einzelworte zerlegen

- ❖ Eine in der Informatik oft gebrauchte Funktion ist die Zerlegung einer Zeichenkette in Teilworte (*tokens*), wobei bestimmte Buchstaben (z.B. Leerzeichen) als Trennzeichen (*delimiters*) gelten.



Wir nehmen dafür eine vordefinierte Java-Klasse: StringTokenizer

- ❖ **StringTokenizer** zerlegt eine Zeichenkette in Teilsegmente (*tokens*)



Enumeration ist in Java eine vordefinierte Schnittstelle

- ❖ Die Java-Schnittstelle **Enumeration** erlaubt es, auf eine beliebige Menge von Elementen (in einem Baum, einer verketteten Liste, ...) seriell zuzugreifen. Die Schnittstelle definiert die Signatur von zwei Methoden:
- ❖ **Test auf Existenz weiterer Elemente:**
 - **public boolean hasMoreElements()**
Ergibt **true**, wenn die Menge weitere Elemente hat, sonst **false**.
- ❖ **Iterator durch die Menge:**
 - **public Object nextElement()**
Ergibt als Resultat ein Element vom Typ **Object**.
Natürlich muss man den Basis-Typ der Elemente wissen, um die Typkonvertierung machen zu können.
- ❖ **Wichtige Einschränkungen von Enumeration:**
 - Der Iterator geht durch jedes Element genau einmal durch. Man kann nicht an den Anfang oder auch nur ein Element zurückgehen.
 - Eine Enumeration garantiert nicht, dass der Iterator die Elemente in sortierter Reihenfolge hervorbringt.

API von StringTokenizer

```
public class StringTokenizer implements Enumeration {
    public StringTokenizer( String str, String delim );
    public StringTokenizer( String str );

    public int countTokens();
    public boolean hasMoreTokens();
    public String nextToken();
    public String nextToken( String delim );

    // Enumeration interface
    public boolean hasMoreElements();
    public Object nextElement();
}
```

Ein Java-Programm ist ein System von Subsystemen

- ❖ **StringTokenizer** und **Enumeration** sind weitere Beispiele von vordefinierten Java-Klassen bzw. -Interfaces
- ❖ Java-Klassen in einem Java-Programm sind in Subsystemen organisiert.
 - Ein Subsystem ist eine Menge von Java-Klassen (*Java class library*)
 - Die Subsysteme in Java heißen Pakete (*packages*)
 - Java 1.1 besteht aus 23 vordefinierten Paketen:

- ❖ java.applet
- ❖ java.awt, java.awt.datatransfer, java.awt.event, java.awt.peer
- ❖ java.beans
- ❖ java.io
- ❖ **java.lang**, java.lang.reflect
- ❖ java.math

- ❖ java.net
- ❖ java.rmi, java.rmi.dgc, java.rmi.registry, java.rmi.server
- ❖ java.security, java.security.acl, java.security.interfaces
- ❖ java.sql
- ❖ java.text
- ❖ **java.util**, java.util.zip
- ❖ **default**

package-Anweisung, Sichtbarkeit in Paketen

- ❖ Ein Java-Programm kann mit einer **package**-Anweisung beginnen.
 - **package mySpecialInfoIPackage;**
- ❖ Wenn die **package**-Anweisung fehlt, ist der Code im Programm automatisch Teil eines namentlich nicht genannten **default**-Paketes.
- ❖ Eine in einem Paket öffentlich (**public**) deklarierte Klasse ist auch aus einem anderen Paket heraus zugreifbar.
 - Eine nicht mit **public** deklarierte Klasse ist außerhalb des Paketes nicht zugreifbar.
- ❖ Alle Mitglieder einer Klasse sind von einer anderen Klasse innerhalb desselben Paketes zugreifbar, wenn sie nicht privat (**private**) deklariert sind.

Java hat ein Internet-weites Namensschema für Bezeichner

- ❖ Wenn man eine Klasse **K** aus einem Paket **p** verwenden will, benutzt man den *voll qualifizierten Namen*, bestehend aus dem Paketnamen und dem Klassennamen:

– **p.K**

- ❖ Beispiel: Die Klasse **StringTokenizer** ist Teil vom Paket **java.util**. Um die Methode **countTokens()** eines Objekts **s** vom Typ **StringTokenizer** aufzurufen, schreibt man:

```
java.util.StringTokenizer s =  
new java.util.StringTokenizer("Zerlegungstext");  
int tokens = s.countTokens();
```

import-Anweisung

- ❖ Man kann lange qualifizierte Namen vermeiden, in dem man das Paket oder die Klasse **importiert**.
- ❖ Die **import**-Anweisung macht Java-Pakete und Java-Klassen für die derzeitige Klasse verfügbar, ohne dass man voll qualifizierte Namen verwenden muss. Es existieren zwei Varianten:
 - **import paket.Klasse;**
 - Die Klasse mit dem Namen **Klasse** im Paket mit dem Namen **paket** ist allein durch ihren Namen bekannt.
 - Beispiel: **import java.util.StringTokenizer;**
 - **import paket.*;**
 - Alle Klassen im Paket mit dem Namen **paket** können mit ihren Klassennamen allein erreicht werden.
 - Beispiel: **import java.util.*;**

import-Anweisung

- ❖ Ein Java-Programm kann beliebig viele **import**-Anweisungen enthalten. Sie müssen allerdings alle am Anfang des Programms (nach der optionalen **package**-Anweisung) erscheinen. Beispiel:

- **import java.util.*;**

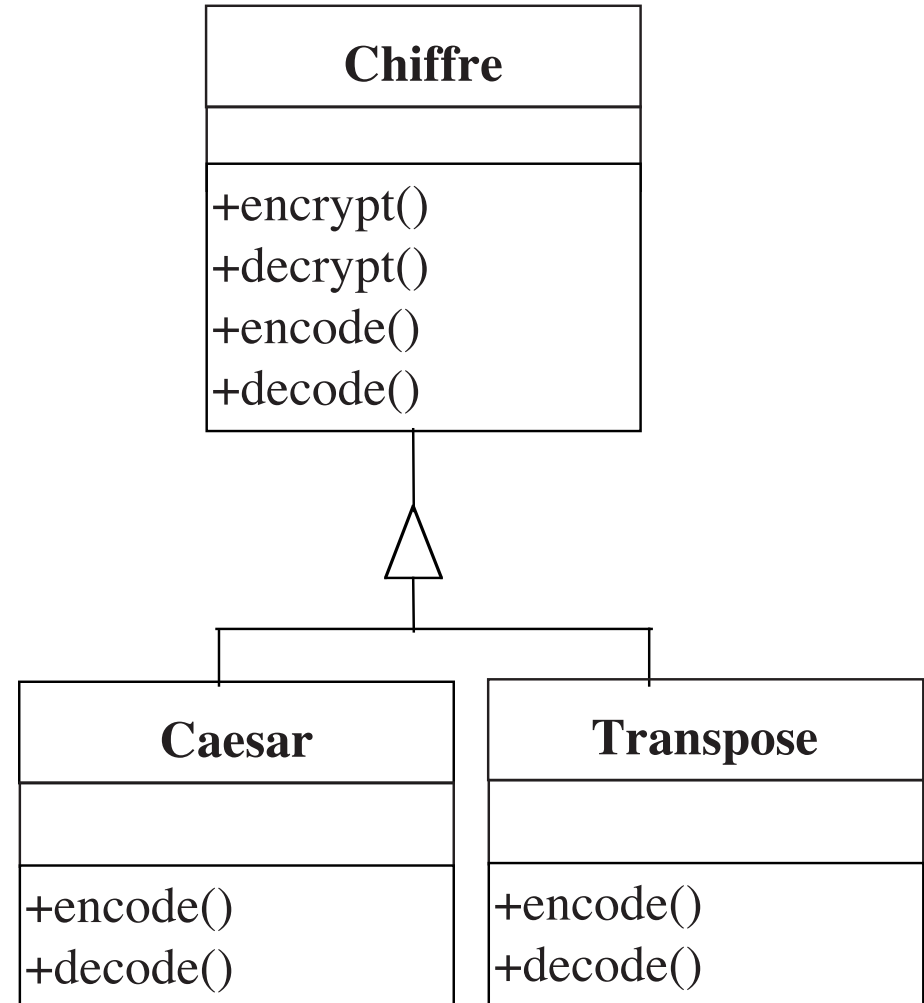
- ❖ Die **import**-Anweisung

- **import java.lang.*;**

wird übrigens implizit am Anfang jedes Java-Programms automatisch vom Java-Compiler eingefügt.

*Jetzt kann die Implementation losgehen.
Der Objekt-Entwurf hatte ergeben:*

- ❖ **Chiffre** als Superklasse mit 4 Methoden:
 - **encrypt ()** verschlüsselt einen Text bestehend aus Wörtern.
 - **decrypt ()** entschlüsselt einen Text bestehend aus Wörtern.
 - **encode ()** wendet einen spezifischen Algorithmus zur Verschlüsselung eines Wortes an.
 - **decode ()** wendet einen spezifischen Algorithmus zur Entschlüsselung eines Wortes an.



Java-Implementation der Chiffre-Superklasse

```
package KryptoBruegge;
import java.util.*;
public abstract class Chiffre {
    public String encrypt(String s) {
        String result = new String("");
        StringTokenizer words = new StringTokenizer(s); // Segmentiere s in Worte
        while (words.hasMoreTokens()) { // Für jedes Wort w in s:
            result = result + " " + encode(words.nextToken()); // Verschlüssele w
        }
        return result;
    } // encrypt()

    public String decrypt(String s) {
        String result = new String("");
        StringTokenizer words = new StringTokenizer(s);
        while (words.hasMoreTokens()) { // Für jedes Wort w in s:
            result = result + " " + decode(words.nextToken()); // Entschlüssele w
        }
        return result;
    } // decrypt()

    public abstract String encode(String word);
    public abstract String decode(String word);
} // Chiffre
```

Die encrypt- and decrypt-Methoden sind für jede Chiffre gleich: Sie sind deshalb voll implementiert.

Abstrakte Methoden haben keinen Rumpf, d.h. sie sind nicht implementiert.

Regeln für abstrakte Methoden und Klassen

- ❖ Eine Klasse, die eine abstrakte Methode enthält, muss als abstrakte Klasse deklariert werden:
 - **public abstract class Chiffre {**
- ❖ Eine abstrakte Klasse kann nicht instantiiert werden. Zur Instantiierung muss eine Unterklasse verwendet werden.
- ❖ Die Unterklasse einer abstrakten Klasse kann nur dann instantiiert werden, wenn sie alle abstrakten Methoden der Superklasse implementiert.
 - Eine Unterklasse braucht nicht alle abstrakten Methoden zu implementieren. Dann muss sie allerdings selber auch wieder als **abstract** deklariert werden.
- ❖ Eine Klasse kann man als **abstract** deklarieren, auch wenn sie keine abstrakten Methoden enthält. Sie könnte z.B. Instanzvariablen enthalten, die für alle Unterklassen gleich sein sollen.

Algorithmus: Caesar-Verschlüsselung

Normales Alphabet: abcdefghijklmnopqrstuvwxyz

Substitutionsalphabet: defghijklmnopqrstuvwxyzabc

- ❖ Implementation der Verschiebung um 3 Buchstaben:

```
ch = (char) ('a' + ((ch - 'a' + 3) % 26));    // "Caesarverschiebung"
```

- ❖ Beispiel: Der Caesar-Code von 'y' ist 'b'

```
(char) ('a' + ((ch - 'a' + 3) % 26))
(char) ('a' + (('y' - 'a' + 3) % 26))    // Substitution 'y' für ch
(char) (97 + ((121 - 97 + 3) % 26))    // Einsetzen der Unicodes für 'a' und 'y'
(char) (97 + (27 % 26))                // Verschieben um 3: Addiere 3
(char) (97 + 1)                         // bilde Resultat wieder auf 'a-z' ab (modulo 26)
(char) (98)                             // konvertiere Ganzzahl (int) in Zeichen (char)
'b'                                     // 98 ist der Unicode von 'b'
```

Implementation der Caesar-Klasse

Caesar erbt von Chiffre, wir müssen also encode() und decode() implementieren

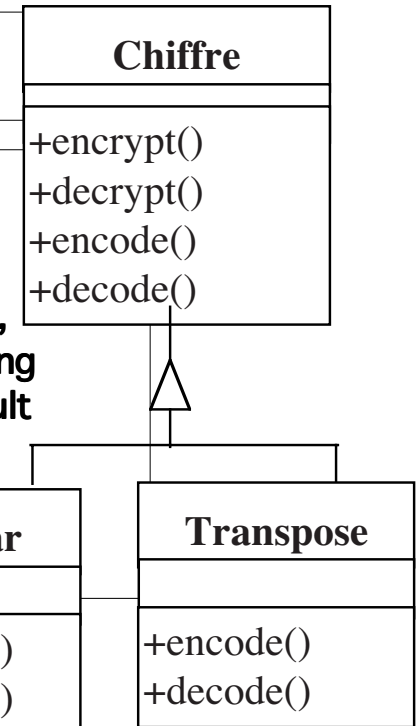
```
public class Caesar extends Chiffre {
```

```
public String encode(String word) {
    String result = new String();
    for (int k = 0; k < word.length(); k++) {
        char ch = word.charAt(k);
        ch = (char) ('a' + ((ch - 'a' + 3) % 26));
        result = result + ch;
    }
    return result;
} // encode()
```

// wir wandern durch das Wort,
// holen uns Zeichen für Zeichen,
// machen die Caesarverschiebung
// und konkatenieren es mit result

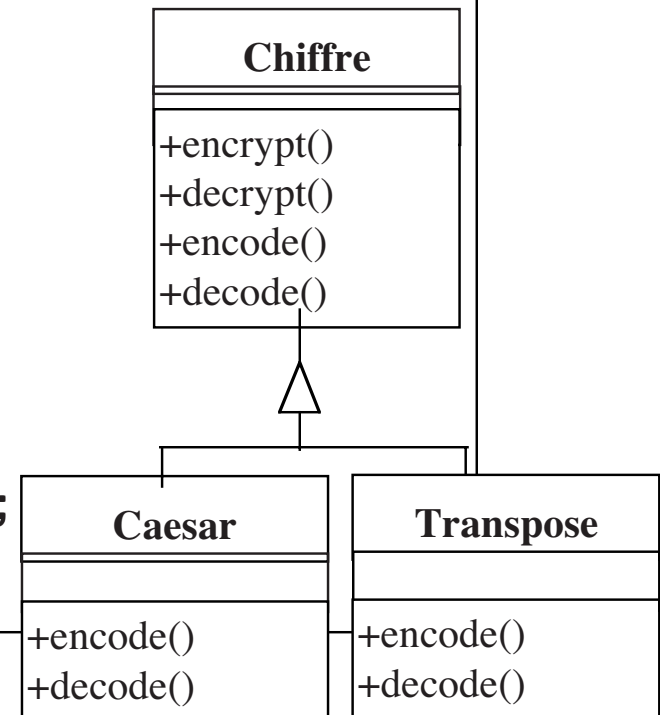
```
public String decode(String word) {
    String result = new String();
    for (int k = 0; k < word.length(); k++) {
        char ch = word.charAt(k);
        ch = (char) ('a' + ((ch - 'a' + 23) % 26));
        result = result + ch;
    }
    return result;
} // decode()
} // Caesar
```

// für jedes Zeichen
// machen wir die umgekehrte
// Caesarverschiebung
// und konkatenieren es mit result



Implementation der Transpose-Klasse

```
public class Transpose extends Chiffre {  
  
    private String reverse(String word) {  
        String result = "";  
        for (int i=word.length()-1; i>=0; i--)  
            result = result + word.charAt(i);  
        return result;  
    }  
  
    public String encode(String word) {  
        return reverse(word);  
    } // encode()  
  
    public String decode(String word) {  
        return reverse(word); // oder: return encode(word);  
    } // decode  
} // Transpose
```



Testprogramm zum Testen der Klassen Caesar und Transpose

encrypt ist eine polymorphe Methode

```
public class Test {
    public static void main(String[] argv) {
        Caesar caesar = new Caesar();
        String plain = "this is the secret message"; // Hier ist die Nachricht
        String secret = caesar.encrypt(plain); // Verschlüssele sie
        System.out.println(" ***** Caesar Chiffre *****");
        System.out.println("Normal: " + plain); // Zeige das Resultat
        System.out.println("Verschlüsselt: " + secret);
        System.out.println("Entschlüsselt: " + caesar.decrypt(secret)); // Entschlüsselung

        Transpose transpose = new Transpose();
        secret = transpose.encrypt(plain);
        System.out.println("\n ***** Transpose Chiffre *****");
        System.out.println("Normal: " + plain); // Zeige das Resultat
        System.out.println("Verschlüsselt: " + secret);
        System.out.println("Entschlüsselt: " + transpose.decrypt(secret));
    } // main()
} // Test
```

Polymorphismus

- ❖ **Polymorphismus** (griechisch): Viele (*poly*) Formen (*morph*).
- ❖ Eine polymorphe Methode hat denselben Namen in verschiedenen Klassen, aber unterschiedliches Verhalten für die jeweiligen Klassen. Beispiele:
 - Die Methode **encrypt ()** der Klassen **Caesar** und **Transpose** ist polymorph.
 - Die Methode **decrypt ()** ist ebenfalls polymorph.

Polymorphe Methoden

- ❖ Die Methoden **encrypt ()** und **decrypt ()** sind polymorphe Methoden.
 - Sie haben dieselben Namen für die Unterklassen **Caesar** und **Transpose**.
 - Je nach Unterklasse haben sie eine völlig unterschiedliche Implementation.
- ❖ Weiß der Compiler immer, welche Methode aufgerufen werden muss?
 - In unserem Fall ja, weil die Methode mit dem Namen der Unterklasse qualifiziert ist.
- ❖ **Statische Bindung** liegt vor, wenn der Compiler zur Compilationszeit entscheiden kann, welche Implementation einer polymorphen Methode aufzurufen ist.

```
public class Test {
    public static void main(String[] argv) {
        Caesar caesar = new Caesar();
        .....
        String secret = caesar.encrypt(plain);
        .....
        System.out.println("Entschlüsselt: "
+ caesar.decrypt(secret));

        Transpose transpose = new Transpose();
        secret = transpose.encrypt(plain);
        .....
        System.out.println("Entschlüsselt: "
+ transpose.decrypt(secret));
    } // main()
} // Test
```

Dynamische Bindung von polymorphen Methoden

- ❖ In diesem Beispiel kann der Compiler nicht zur Compilationszeit entscheiden, ob bei **message.encrypt()** die **Caesar**-Implementation oder die **Transpose**-Implementation der Methode **encrypt()** aufgerufen werden muss. Dasselbe gilt für **message.decrypt()**.
- ❖ Welche Implementation gewählt wird, hängt davon ab, ob das von **message** referenzierte Objekt vom Typ **Caesar** oder vom Typ **Transpose** ist.
- ❖ **Dynamische Bindung** liegt vor, wenn die Entscheidung, welche Implementation einer polymorphen Methoden aufzurufen ist, erst zur Laufzeit erfolgen kann.

```
public class Test {  
    public static void main(String[] argv) {  
        Chiffre message = new Caesar();  
        .....  
        String secret = message.encrypt(plain);  
        .....  
        System.out.println(message.decrypt(secret));  
  
        message = new Transpose();  
        .....  
        secret = message.encrypt(plain);  
        .....  
        System.out.println(message.decrypt(secret));  
    } // main()  
} // Test
```

Ausgabe des Testprogramms

Kryptotest 1

```
***** Caesar Chiffre *****  
Normal: dies ist eine geheime nachricht  
Verschlüsselt: glhv lvw hlqh jkhlph qdfkulfkw  
Entschlüsselt: dies ist eine geheime nachricht  
***** Transpose Chiffre *****  
Normal: dies ist eine geheime nachricht  
Verschlüsselt: seid tsi enie emieheg thcirhcan  
Entschlüsselt: dies ist eine geheime nachricht
```

**Welche Ausgabe kriegen wir für:
"Dies ist eine !@\$ geheime Nachricht?"**

```
***** Caesar Chiffre *****
Normal: Dies ist eine !@$ geheime Nachricht
Verschlüsselt: alhv lvw hlqh X] Z[ jhkhlp Qdfkulfkw
Entschlüsselt: xies ist eine otqr geheime hachricht
***** Transpose Chiffre *****
Normal: Dies ist eine !@$ geheime Nachricht
Verschlüsselt: seiD tsi enie $#@! emieheg thcirhcaN
Entschlüsselt: Dies ist eine !@$ geheime Nachricht
```

Was machen wir jetzt?

Zur Wiederholung: ASCII-Zeichensatz

Zeichen	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
Code	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Zeichen	0	1	2	3	4	5	6	7	8	9						
Code	48	49	50	51	52	53	54	55	56	57						
Zeichen	:	;	<	=	>	?	@									
Code	58	59	60	61	62	63	64									
Zeichen	A	B	C	D	E	F	G	H	I	J	K	L	M			
Code	65	66	67	68	69	70	71	72	73	74	75	76	77			
Zeichen	N	O	P	Q	R	S	T	U	V	W	X	Y	Z			
Code	78	79	80	81	82	83	84	85	86	87	88	89	90			
Zeichen	[\]	^												
Code	91	92	93	94	95	96										
Zeichen	a	b	c	d	e	f	g	h	i	j	k	l	m			
Code	97	98	99	100	101	102	103	104	105	106	107	108	109			
Zeichen	n	o	p	q	r	s	t	u	v	w	x	y	z			
Code	110	111	112	113	114	115	116	117	118	119	120	121	122			
Zeichen	{		}	~												
Code	123	124	125	126												

Erweiterung des Alphabets auf ASCII-Zeichen 32 bis 126

```
ch = (char) ('a' + ((ch - 'a' + 3) % 26));
```

Mit expliziter Typkonvertierung:

```
ch = (char)((int)'a' + ((int) ch -(int) 'a'+ 3) %  
26);
```

- ❖ Erweitertes Alphabet (Ascii-Codes 32 bis 126):

```
SP!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN  
OPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxy  
z{|}~
```

- ❖ Substitutionsalphabet:

```
#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMN  
OPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxy  
z{|}~SP!"
```

```
ch = (char) (' ' + ((ch - ' ' + 3) % 98)); // "Caesarverschiebung" in encode
```

```
ch = (char) (' ' + ((ch - ' ' + 95) % 98)) // "Caesarverschiebung" in decode
```

Die neue Klasse CaesarAscii

```
import KryptoBruegge.*;
public class CaesarAscii extends Chiffre {
public String encode(String word) {
    String result = new String();
    for (int k = 0; k < word.length(); k++) {
        char ch = word.charAt(k);
        ch = (char) (' ' + ((ch - ' ' + 3) % 98));
        result = result + ch;
    }
    return result;
} // encode()
public String decode(String word) {
    String result = new String();
    for (int k = 0; k < word.length(); k++) {
        char ch = word.charAt(k);
        ch = (char) (' ' + ((ch - ' ' + 95) % 98));
        result = result + ch;
    }
    return result;
} // decode()
} // Caesar
```

Kryptotest 2

Das modifizierte Testprogramm

```
import KryptoBruegge.*;
public class Test {
    public static void main(String[] argv) {
        Caesar caesar = new Caesar();
        String plain = "Dies ist eine geheime Nachricht";
        String secret = caesar.encrypt(plain);
        System.out.println(" ***** Caesar Chiffre *****");
        System.out.println("Normal: " + plain);
        System.out.println("Verschlüsselt: " + secret);
        System.out.println("Entschlüsselt: " + caesar.decrypt(secret));
        Transpose transpose = new Transpose();
        secret = transpose.encrypt(plain);
        System.out.println("\n ***** Transpose Chiffre *****");
        System.out.println("Normal: " + plain);
        System.out.println("Verschlüsselt: " + secret);
        System.out.println("Entschlüsselt: " + transpose.decrypt(secret));
        CaesarAscii caesarascii = new CaesarAscii();
        secret = caesarascii.encrypt(plain);
        System.out.println("\n ***** CaesarAscii Chiffre *****");
        System.out.println("Normal: " + plain);
        System.out.println("Verschlüsselt: " + secret);
        System.out.println("Entschlüsselt: " + caesarascii.decrypt(secret));
    } // main()
} // Test
```



Was wir haben

- ❖ Die Aufrufe der Verschlüsselung und Entschlüsselungsmethoden sind abhängig vom gewählten Algorithmus.
- ❖ Die Einführung eines solchen neuen Algorithmus erfordert in der Wirklichkeit viele Änderungen und viele Recompilationen

....

```
Caesar caesar = new Caesar();  
String secret = caesar.encrypt(plain);  
System.out.println(caesar.decrypt(secret);
```

....

```
Transpose transpose = new Transpose();  
secret = transpose.encrypt(plain);  
System.out.println(transpose.decrypt(secret);
```

.....

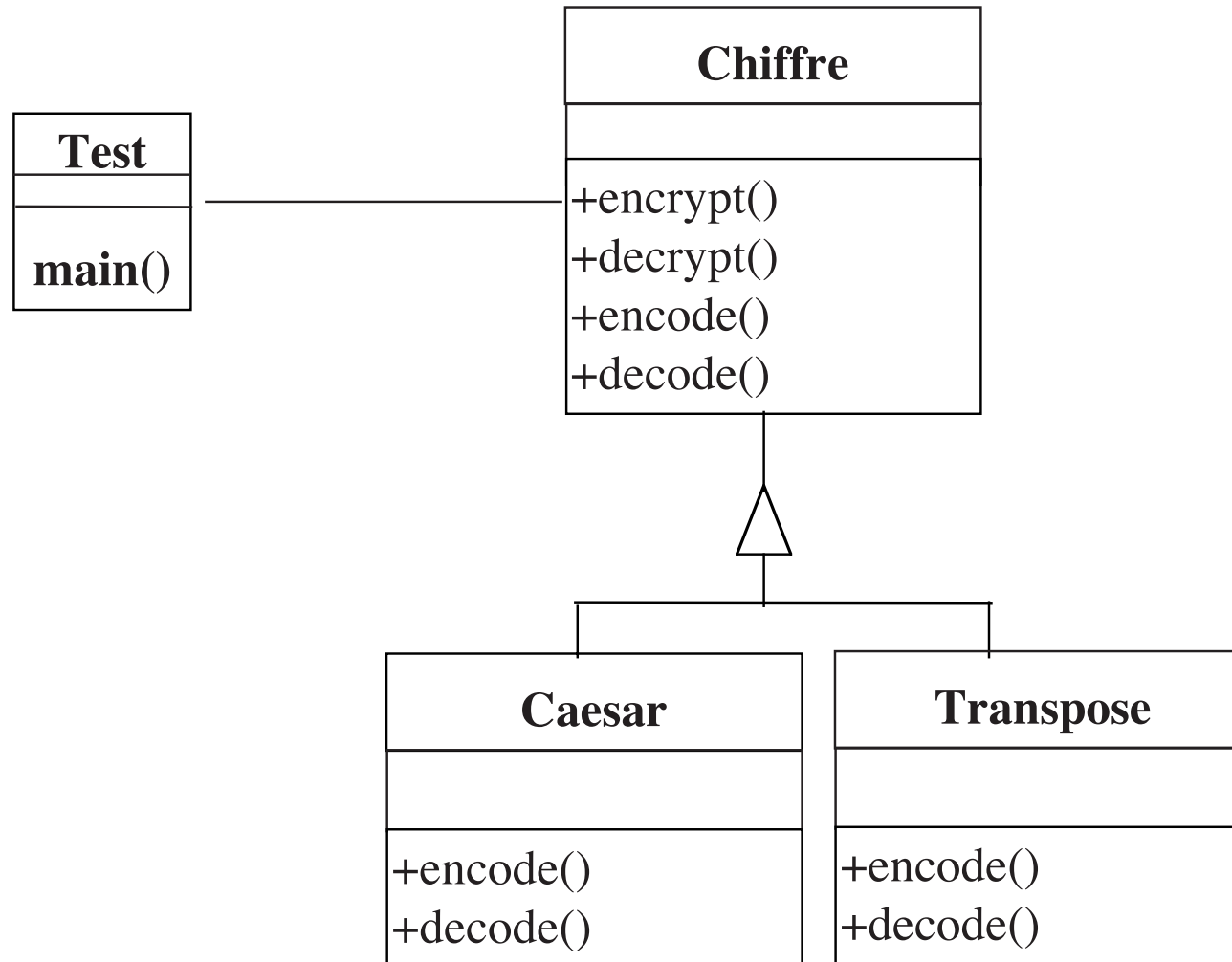
```
CaesarAscii caesarascii = new CaesarAscii();  
secret = caesarascii.encrypt(plain);  
System.out.println(caesarascii.decrypt(secret)
```

....

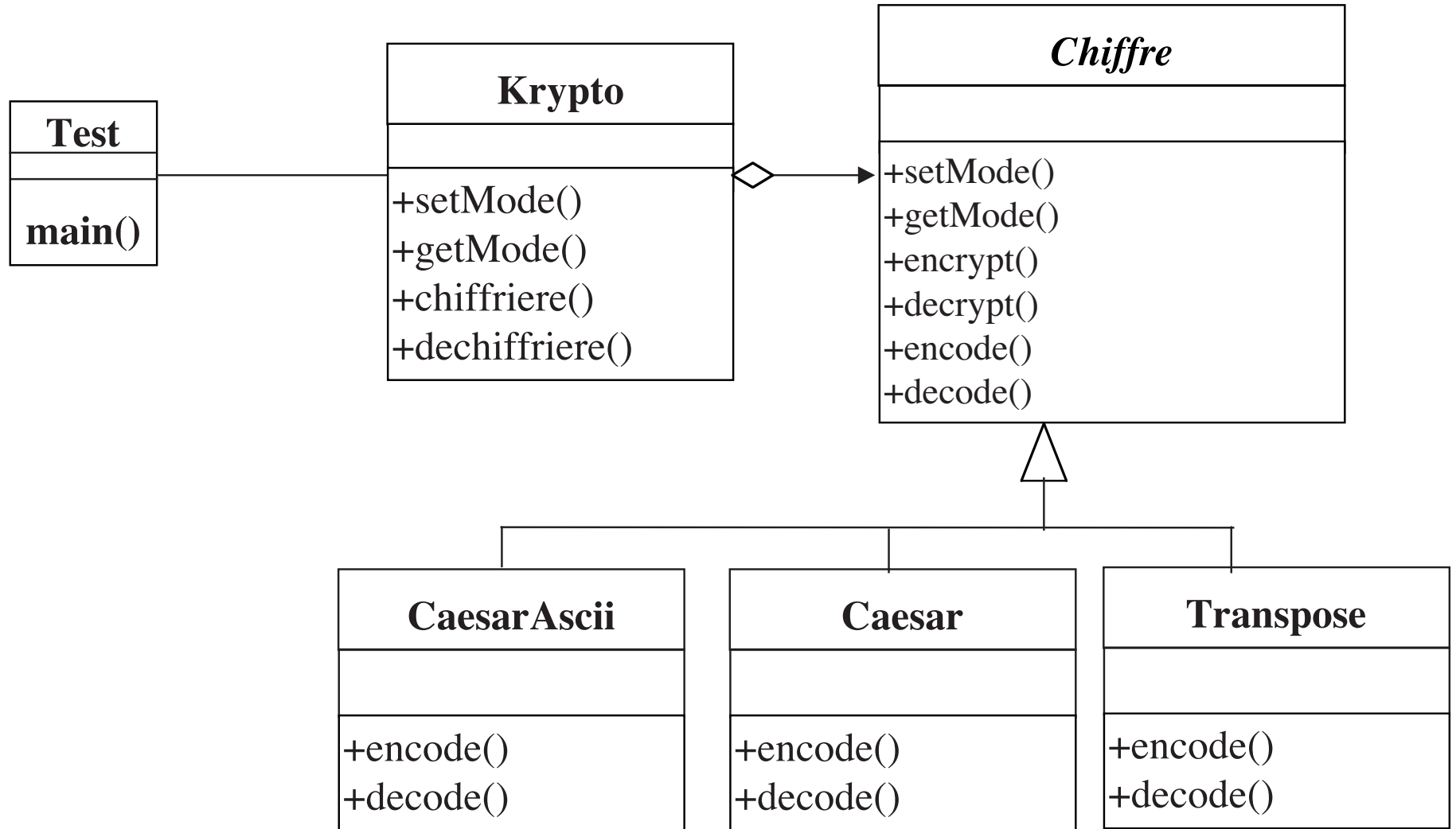
Was wir wirklich wollen

- ❖ Benutzer legt sich auf einen bestimmten Chiffrierungsalgorithmus fest, z.B. Caesar, durch Aufrufen einer Methode **setMode**:
 - **setMode ("Caesar") ;**
- ❖ Für gegebene Nachrichten ruft der Benutzer die Verschlüsselungs- und Entschlüsselungsmethoden auf, ohne sich darum zu kümmern müssen, welcher Algorithmus eigentlich benutzt wird.
 - **...message.chiffriere(s) ;**
 - **...message.dechiffriere(s) ;**
- ❖ Bei der Einführung eines neuen Chiffrieralgorithmus gibt es 2 Fälle:
 - Benutzer, die nicht umstellen wollen,
 - brauchen nichts zu tun!
 - Benutzer, die den neuen Algorithmus verwenden wollen,
 - müssen wissen, mit welchem Argument die Methode **setMode** aufzurufen ist, und die Stelle im Programm ändern, wo der Modus gesetzt wird: **setMode ("CaesarAscii") ;**

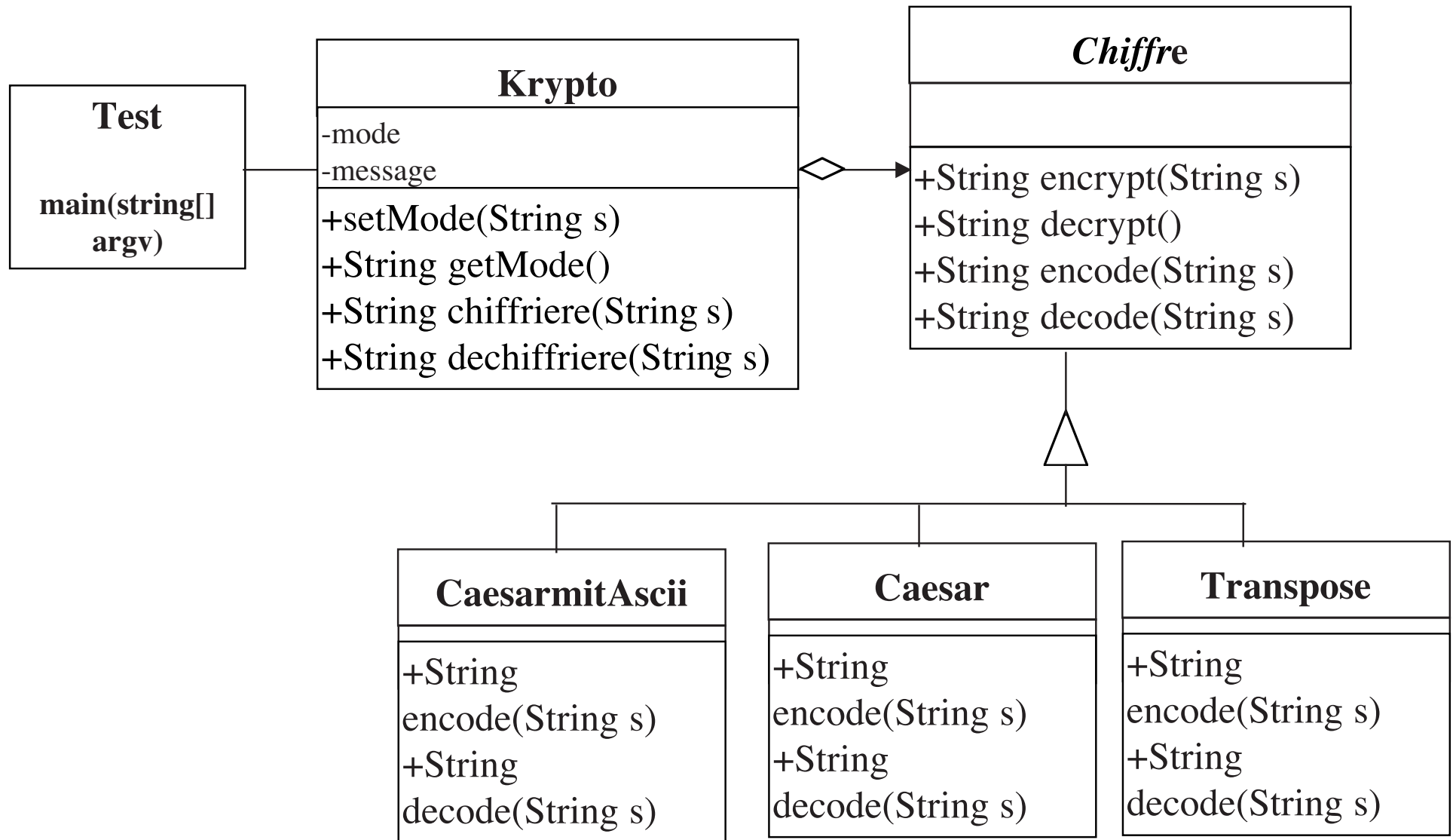
Bisheriges Analyse-Modell



Besseres Analyse-Modell



Objekt-Modell mit Schnittstellen



Java-Implementation der neuen Klasse Krypto

```
public class Krypto {
    private Chiffre message, caesar, transpose, caesarascii;
    private String kryptoMode;

    public Krypto () {
        caesar = new Caesar();
        transpose = new Transpose();
        caesarascii = new CaesarAscii();
        message = caesar;
    }
    public void setMode (String s) {
        // ...
    }
    public String getMode () {
        // ...
    }
    public String chiffriere (String s) { return message.encrypt(s); }
    public String dechiffriere (String s) { return message.decrypt(s); }
} // Krypto
```

Beispiel: Java-Implementation mit Krypto

```
public class Kryptotest3 {  
    public static void main(String[] argv) {
```

Code (krypto3)

```
// Festlegung des Algorithmus:
```

```
    Krypto message = new Krypto();  
    message.setMode("Transpose");
```

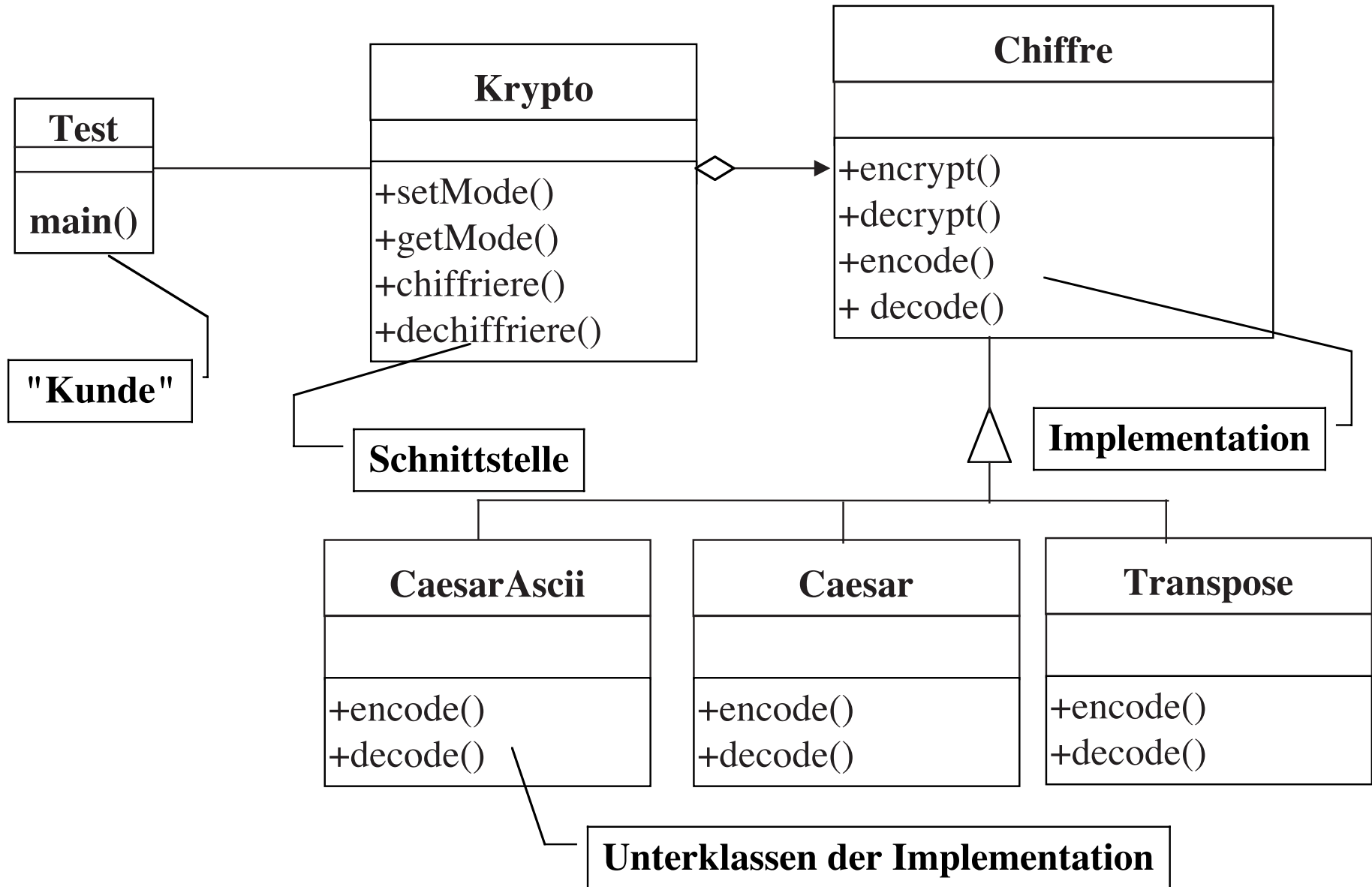
```
//Anwendung des Algorithmus:
```

```
    String plain = "Dies ist eine geheime Nachricht";  
    String secret = message.chiffriere(plain);  
    System.out.println("***** Chiffre: " + message.getMode() + " *****");  
    System.out.println("Normal: " + plain);  
    System.out.println("Verschlüsselt: " + secret);  
    System.out.println("Entschlüsselt: " + message.dechiffriere(secret));
```

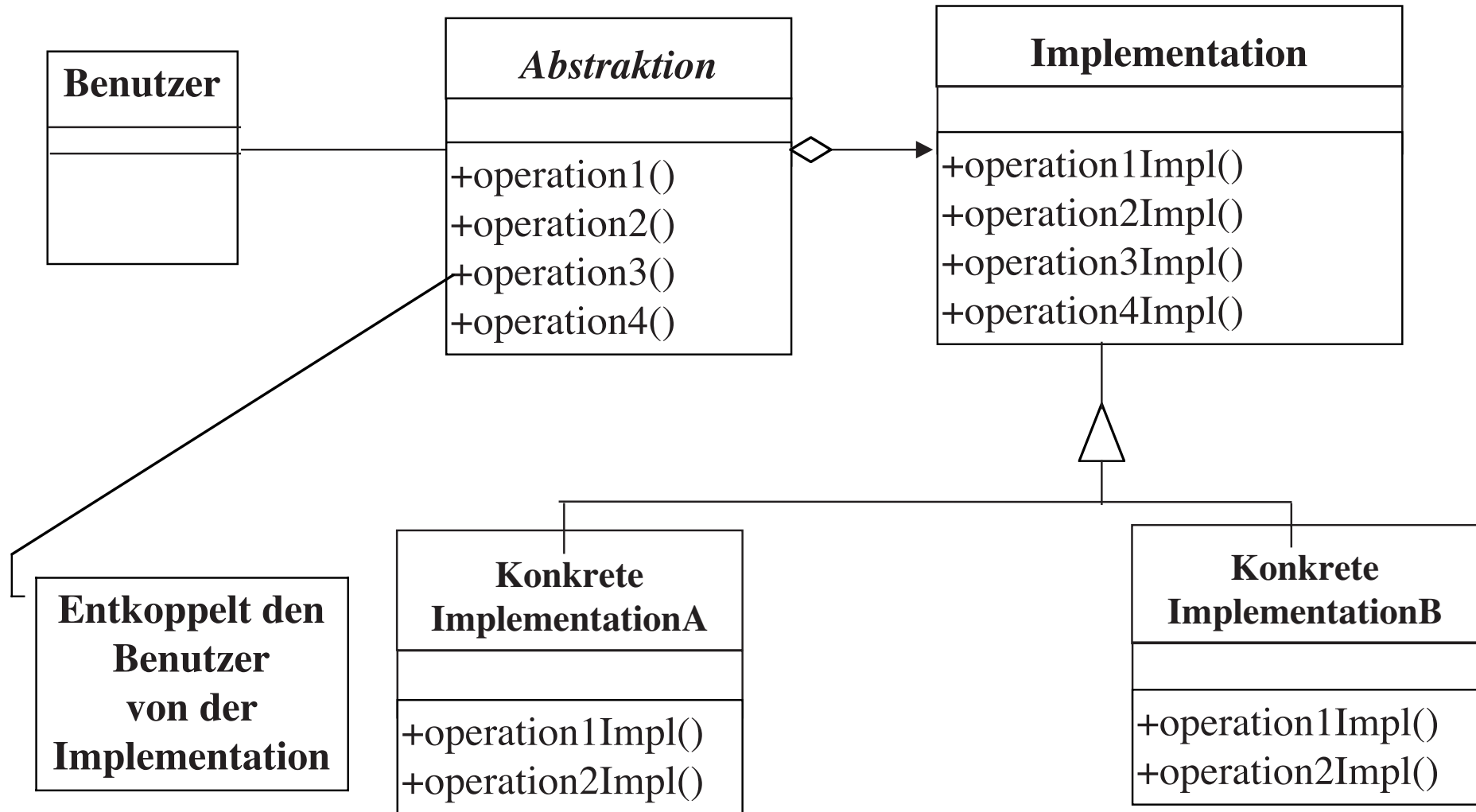
```
    } // main()
```

```
} // Test
```

Schauen wir uns noch einmal das Modell an



Ein wichtiges Entwurfsmuster: Die Brücke



Anwendungsmöglichkeiten für das Brückenmuster

- ❖ **Flexibilität:** Wenn man eine permanente Bindung zwischen der Schnittstelle und ihrer Implementation vermeiden will.
 - Wenn die Implementation zur Laufzeit selektiert werden soll
 - Wenn zwischen mehreren Implementationen zur Laufzeit hin- und hergewechselt werden soll.
 - Änderungen in der Implementation sollen vom Benutzer nicht bemerkt werden, d.h. der Benutzercode muss nicht recompiliert werden.

- ❖ **Erweiterbarkeit:**
 - Die Schnittstelle soll mit einer bestimmten Implementation geliefert werden.
 - Später soll eine neue Implementation ohne große Änderungen im Benutzercode hinzugefügt werden können.

Was noch zu tun ist

Unser Anwendungsbeispiel benutzt eine noch nicht ganz "reine" Brücke

```
public class Krypto {  
  
    private Chiffre message, caesar,  
        transpose, caesarascii;  
    private String kryptoMode;  
  
    public Krypto() {  
        caesar = new Caesar();  
        transpose = new Transpose();  
        caesarascii = new CaesarAscii();  
        message = caesar;  
    }  
  
    ....  
} // Krypto
```

```
public class Krypto {  
  
    public Krypto() {  
    }  
  
    ....  
} // Krypto
```

```
public abstract class Chiffre {  
    private Chiffre message, caesar,  
        transpose, caesarascii;  
    private String kryptoMode;  
  
    public Chiffre() {  
        caesar = new Caesar();  
        transpose = new Transpose();  
        caesarascii = new CaesarAscii();  
        message = caesar;  
    }  
  
    ....  
} // Chiffre
```

Das Beispiel machen wir in Info II weiter

Preisaufgabe

- ❖ Bereinigen Sie den Code für **Krypto**
 - Verschieben sie alle Felder und Methodenimplementationen aus der Klasse **Krypto** in die Klasse **Chiffre**.
- ❖ Begründen Sie, warum **Krypto** keine abstrakte Klasse sein kann.
- ❖ Erstellen Sie einen verbesserten Caesar-Algorithmus mit dem Schlüsselwort "InfoListSpitze!".
 - Modellieren und implementieren Sie Ihren Algorithmus
- ❖ Der neue Algorithmus muss für eine beliebige, Ihnen unbekannte verschlüsselte Zeichenkette funktionieren, die als Kommandozeilen-Argument gelesen wird.
- ❖ Senden Sie ein:
 - Das verbesserte **Krypto**-Paket.
 - Ihr Testprogramm
 - Protokoll eines Testlaufs (Testdaten)
- ❖ Abgabetermin: 15. März 2001

Was muß ich als Informatiker können?

❖ **Probleme lösen können**

- Eine Problembeschreibung in ein Modell übersetzen können
- Das Modell implementieren können
- Basiswissen anwenden: Theorie

❖ **Reden und Schreiben**

- Mit einem Anwender reden können
- Teamarbeit: Mit meinen Mit-Entwicklern reden können
- Eine neue Idee präsentieren können

❖ **Technologie- "freudig" sein**

- Halbwertszeit des Informatikwissens ist 2-3 Jahre

Das wars!

- ❖ Vielen Dank fürs Zuhören
- ❖ Wir sehen uns wieder am 23. April 2001 in Info II.