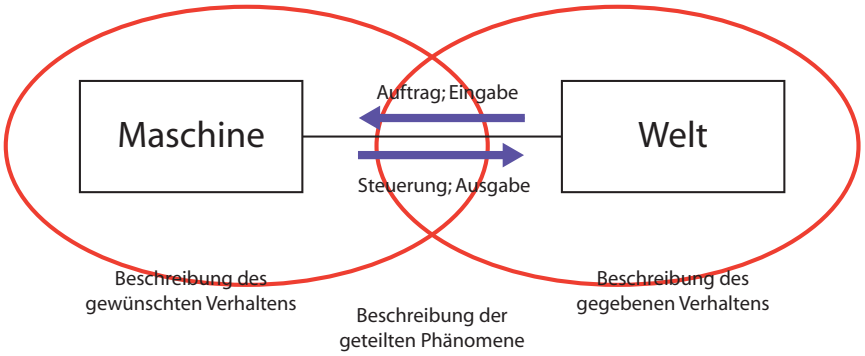
Softwareabstraktionen

Beschreibungen und Abstraktion

Nochmals: Was ist Softwaretechnik?

Die Definitionen der IEEE und von Helmut Balzert betonen das *Ingenieursmäßige* der Softwareentwicklung – man kann aber auch fragen: was ist das Charakteristische *der Softwareentwicklung* als Ingenieurstätigkeit? (Vielleicht bekommt man dann auch eine Definition von Softwaretechnik, die ihre *Besonderheit* besser erfasst.)

Was ist die Besonderheit von Software? Betrachten wir dazu folgende Abbildung



- In der *Welt* sind die Aufgaben, Probleme, die man mit Einsatz von Computern lösen möchte.
- Mittel dafür ist die *Maschine*, in der Regel eine universelle, abstrakte Maschine, die Symbole und Informationen verarbeiten kann, aber nicht auf einen bestimmten Zweck festgelegt ist.
- Die Welt und die Maschine stehen in einer Beziehung: Aus der Welt kommt ein Auftrag an die Maschine; aus der Welt kommen Eingaben, die verarbeitet werden sollen. Die Maschine steuert Geräte in der Welt; die Maschine erzeugt Ausgaben, die in der Welt interpretiert und verwendet werden können (je nach Art des Problems und der Software)

Beispiele können etwa sein

- Die Motorsteuerung eines PKW, bei der ein Mikrocontroller eingesetzt wird, der über Sensoren und Aktuatoren mit der Welt verbunden ist. Aus der Welt kommt über einen Sensor ein Auftrag an die Motorsteuerung (eine „Momentenanforderung“, wenn das Gaspedal betätigt wird) und die Motorsteuerung berechnet dann eine *Steuerung*, die über einen Aktuator an den Motor geht.
- Ein Unternehmen wickelt sein Bestell- und Auftragswesen mit einem Computer ab. Eingesetzt werden PCs, die mit einem Server vernetzt sind. Alle in der Welt vorkommenden Informationen über Aufträge usw. werden im Computer als *symbolische Information* repräsentiert. Zu den wirklichen Kunden gibt es Datensätze über Kunden, ebenso zu den Aufträgen usw. Mitarbeiter der Firma verwenden ihre PCs, um diese Informationen an die Maschine zu geben, diese verarbeitet diese Information nach gewissen Regeln (die auch wieder ihren Ursprung in der Welt haben: Abläufe im Unternehmen, Vertragsrecht u.ä.) und erzeugt neue Informationen, die dann wieder von Mitarbeitern (oder anderen Maschinen) verwendet werden. Die Maschine, der Computer ist eine *Simulation* von Dienstleistungen des Unternehmens.

Wie kommt nun Software ins Spiel?

- Einerseits *gibt* es ein bestimmtes Verhalten der Welt. Es ist vorgegeben durch physikalische Gesetze bei der Steuerung, durch kaufmännische Vorgehensweisen beim Bestellwesen.
- Andererseits *benötigt* die Maschine eine Beschreibung, wie sie sich verhalten soll, um von der abstrakten Maschine zur konkreten Maschine zu werden, die einen *bestimmten* Zweck erfüllt.
- Was ist diese Beschreibung? Es ist das *Programm*, das die Maschine steuert – also der *Code*, aus dem das ausführbare Programm erzeugt wird.

Damit diese Beschreibung das Richtige tut, benötigt man in Wahrheit *drei* Beschreibungen

- Beschreibung des gegebenen Verhaltens der Welt
- Beschreibung des gewünschten Verhaltens der Maschine
- Beschreibung der Phänomene, die Maschine und Welt teilen

Wie unterschiedlich diese Beschreibungen sein können, erkennt man, wenn man unsere beiden Beispiele der Motorsteuerung und des Bestellwesens durchdenkt: Was ist der Inhalt der jeweils benötigten Beschreibungen?

Fassen wir die Diskussion mit einem Zitat zusammen:

> Our business is engineering – making machines to serve useful purposes in the world.
> Software development is engineering in a second sense: it is the engineering of complex structures of descriptions. An engineer confronted with the problem of bridging a river designs and builds a structure of parts, using concrete and metal as the raw material. A software developer confronted with the problem of creating a system designs and builds a structure of descriptions, using languages and notations as the raw material.
>
> — Michael Jackson

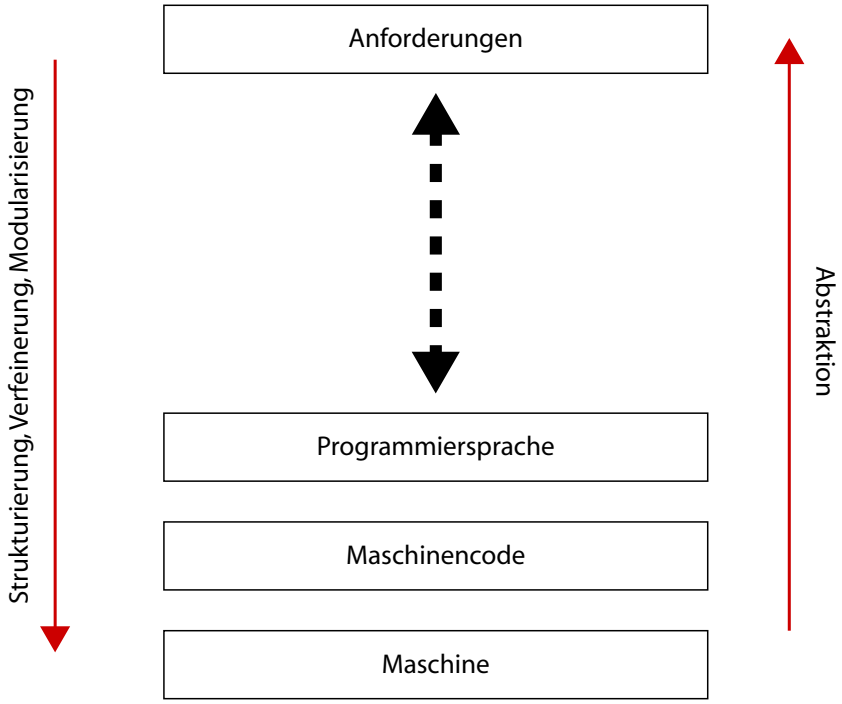
Abstraktion & Strukturierung

Was haben diese Überlegungen nun mit dem Begriff der Abstraktion zu tun?

Wir haben gesehen, dass es die Aufgabe der Softwareentwicklung ist, (1) eine Beschreibung des gewünschten Verhaltens der Maschine, des Computers zu finden, so dass (2) die Phänomene, die die Maschine mit der Welt teilt, zu dem (3) gegebenen Verhalten der Welt gemäß dem intendierten Zweck passen.

Eine Frage von *Sprachen* und *Notationen*. Das Problem ist nur, dass es eine „Sprachlücke“ gibt:

Schlussendlich muss das gewünschte Verhalten der Maschine in einem ausführbaren Programm – also in Maschinencode – vorliegen. Das ist aber eine Sprache, die Welten von einer Sprache entfernt ist, mit der wir das gewünschte Verhalten der Maschine aus Sicht der Welt, aus Sicht der Anforderungen beschreiben würden!



Offensichtlich kann (und muss) man die Beschreibungslücke in zwei Richtungen angehen:

- Von oben nach unten („top down“) durch Strukturierung, Verfeinerung, Modularisierung der Anforderungen, der gewünschten Funktionalität – so dass sie schließlich auch von einem Computer verstanden werden können.
- Von unten nach oben („bottom up“) durch Abstraktion — dadurch dass unwesentliche Details nicht mehr beachtet werden müssen, und nur die wesentlichen Gesichtspunkte beschrieben werden müssen.

Für beide Vorgehensweisen kennen wir alle Beispiele:

- Es ist ein bekanntes Prinzip der Informatik „divide and conquer“, Probleme dadurch zu lösen, dass man sie in einfachere Teilprobleme zerlegt, diese löst und dann die Teillösungen zusammensetzt. Viele Algorithmen sind nach diesem Strickmuster gebaut – etwa rekursive Verfahren.
- Auch die umgekehrte Richtung, die Abstraktion. Heute wird kaum noch in Assembler programmiert, sondern in Programmierspra-

chen wie C++, C# oder Java. Funktions- und Klassenbibliotheken stellen Mechanismen zur Verfügung, über deren Implementierung wir uns keine Gedanken machen müssen (normalerweise jedenfalls nicht). Man kann etwa in C# die Funktionalität der Windows-API nutzen, ohne Details kennen zu müssen, wie Windows tatsächlich arbeitet. (Obwohl es nicht verkehrt ist, zu wissen, was dahinter steckt.)

In dieser Vorlesung werden wir auch beide Richtungen betrachten. Heute zunächst die Richtung der Abstraktion – und zwar die elementarsten Techniken, die wir in der Softwareentwicklung brauchen: Prozedurale Abstraktion und Datenabstraktion.

Später in der Vorlesung werden wir Techniken betrachten, mit denen man „top down“ vorgeht.

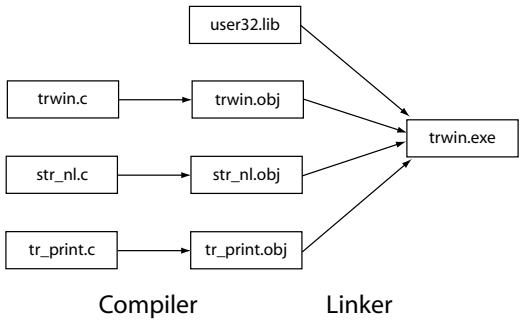
Um Missverständnissen vorzubeugen: es gibt nicht *die* Abstraktionen. Die folgende Diskussion mag diesen Eindruck erwecken, weil wir nur das arg Grundlegende betrachten. Je weiter entfernt wir von der Maschine sind, um so mehr benötigen wir Abstraktionen, die der Sache *angemessen* sind, wegen der wir Software bauen; Abstraktionen also, die aus der wirklichen Welt gewonnen werden müssen, nicht aus der Welt der Maschine und der Informatik!

Doch nun zunächst zum Elementaren:

Prozedurale Abstraktion

Modularisierung und Abstraktion

Die Zerlegung in *Module* hat ihren Ursprung in der getrennten Übersetzung von Programmteilen.



Das kann man verallgemeinern:

Modularisierung ist die Aufteilung von Systemen in Bestandteile, die in hohem Maße unabhängig voneinander sind.

Modularisierung dient der

- Reduktion von Komplexität,
- Parallelisierung, Arbeitsteilung im Systembau,
- Wartbarkeit durch Lokalisierung und
- Wiederverwendbarkeit von Modulen

Modularisierung wird erreicht durch *Abstraktion*. Man spricht auch von *Information Hiding* oder vom *Geheimnisprinzip*.

David L. Parnas 1972: „On the Criteria to Be Used in Decomposing Systems into Modules“

> Every module ... is characterized by its knowledge of a design decision which it hides from all others.
>
> — David Parnas

Abstraktion ist eine Art der Zerlegung durch Wechsel des betrachteten Detailgrads: Man abstrahiert, indem man von (unwesentlichen) Details absieht und dadurch die Sache vereinfacht, aufs Wesentliche konzentriert.

Arten der Abstraktion

- Prozedurale Abstraktion
- Datenabstraktion
- Objektorientierung

Prozedurale Abstraktion

Die Grundidee der prozeduralen Abstraktion:

- Ein Modul kommuniziert mit seiner Umwelt nur über eine definierte *Schnittstelle*
- Der Verwender eines Moduls braucht keinerlei Kennntnis des inneren Aufbaus und der inneren Arbeitsweise des Moduls

- Die Korrektheit des Moduls kann verifiziert werden, indem geprüft wird, ob es seine Schnittstelle erfüllt – also ohne sein Wirken innerhalb eines Gesamtsystems.

Man spricht auch von *Design by Contract* (Bertrand Meyer, Designer von Eiffel)

Die Schnittstelle eines Moduls kann als ein *Vertrag* zwischen einem *Anbieter*, dem Modul, und einem *Verwender* betrachtet werden. Beide Partner gehen einen Vertrag ein:

- Der Verwender verpflichtet sich, eine bestimmte Situation zu schaffen, in der er das Modul verwendet:
  Der Verwender erfüllt eine Vorbedingung.
- Der Anbieter verpflichtet sich, ein bestimmes Ergebnis zu liefern:
  Der Anbieter garantiert eine Nachbedingung.

Es ist möglich durch formales Kalkül zu beweisen, ob eine bestimmte Funktion bei gegebener Vorbedingung ihre Nachbedingung garantieren kann: Hoare-Kalkül.

Ein Beispiel:

```
int findA( int[] a, int val)
{
  for ( int i = 0; i < a.length; i++ )
  {
    if ( a[i] == val )
      return i;
  }
  return a.length;
}

int findB( int[] a, int val )
{
  for ( int i = a.length-1; i >= 0; i-- )
  {
    if ( a[i] == val )
      return i;
  }
  return -1;
}
```

Worin besteht der Unterschied? Wie kann man die Schnittstelle spezifizieren?

Spezifikation der Schnittstelle

Eine Spezifikation einer Funktion/Methode besteht aus:

- der Signatur der Funktion,
- einer Vorbedingung — Schlüsselwort *pre*,
- einem Ergebnis — Schlüsselwort *return*,
- einer Nachbedingung — Schlüsselwort *post*,
- einer Angabe über Zustandsänderungen — Schlüsselwort *modifies*,
- Angaben zu Ausnahmen — Schlüsselwort *throws*

Beispiel:

```
int find( int[] a, int val )
  @pre   : val occures exactly once in a
  @return: int i
  @post  : a[i] = val
```

Man kann Schnittstellen anhand ihrer Eigenschaften analysieren

Minimalität Eine Spezifikation ist minimaler, wenn sie weniger Einschränkungen in Bezug auf das Verhalten der Funktion macht. Man strebt Minimalität an.

Determiniert/Unterdeterminiert Der Unterschied betrifft die Frage, ob für bestimmte Eingabewerte mehr als ein Ergebnis möglich ist.

Determininistisch Eine Implementierung ist deterministisch, wenn sie stets bei gleicher Eingabe dasselbe Ergebnis produziert.

Allgemeinheit Eine Spezifikation ist allgemeiner, wenn sie eine größere Menge von Eingabewerten erlaubt. Man strebt Allgemeinheit an.

deklarativ vs. prozedural Eine Spezifikation ist deklarativ, wenn sie angibt, *was* erreicht werden soll, aber nicht *wie* dieses geschehen soll.

Datenabstraktion

Idee

Die Idee der prozeduralen Abstraktion ist die Trennung der Implementierung einer Funktion von ihrer Verwendung.

Diese Idee kann man auf die Verwendung von Daten und Datenstrukturen erweitern:

Trennung der Art, wie Daten verwendet werden, von der Art, wie sie im Speicher abgelegt und verwaltet werden.

Wie wird das erreicht?

Charakterisieren der Daten durch ihr Verhalten, durch das, was mit ihnen gemacht werden kann – und nicht durch ihre Repräsentation im Speicher des Computers.

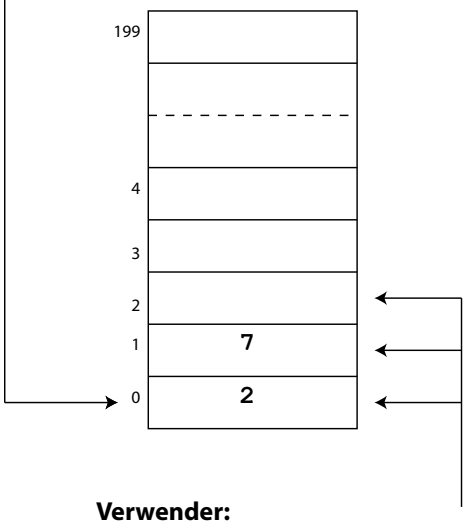
D.h. man hat Methoden zur Verwendung der Daten an Stelle der Kenntnis der Struktur der Daten – das ist Datenabstraktion.

Beispiel eines Stapels

Ohne Datenabstraktion könnte die Beziehung zwischen einem Anbieter eines Stapels und dem Verwender des Stapels (Stack) so aussehen:

Anbieter:

```
int* top = (int*) malloc( 200*sizeof(int) )
```



Verwender:

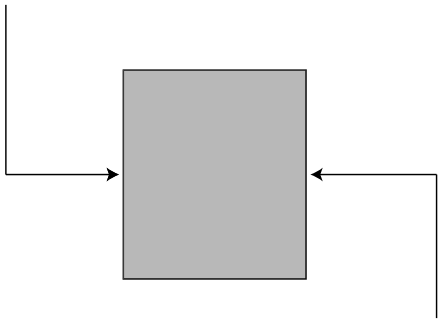
```
*top = 2; top++; // 2 auf den Stack
*top = 7; top++; // 7 auf den Stack
int j = *(top-1);// 7 verwendet
top--;           // 7 vom Stack genommen
```

Es ist offensichtlich, worin der Haken besteht!

Die Alternative sieht schematisch so aus:

Anbieter:

```
stack_type stack
```



Verwender:

```
stack.push( 2 );
stack.push( 7 );
j = stack.top();
stack.pop();
```

Man spricht auch von *gekapseltem Speicher* oder *abstrakten Datentypen*

Die Datenabstraktion ist ein Grundprinzip der Objektorientierung.

Setzt man das Konzept in eine Implementierung in Java um, erhält man folgenden Code – vergleiche dazu auch die Implementierung in der Java API: http://download.oracle.com/javase/7/docs/api/java/util/Stack.html

```
/** ---------------------------------------------------------------------------
 * Implementierung eines (einfachen) Stacks
 * als Musterbeispiel für die Vorlesung Softwaretechnik
 *
 * Copyright (c) 2010 - 2012
 * by Burkhardt Renz Technische Hochschule Mittelhessen.
 *
 * swt-vl is free software; you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free
 * Software Foundation; either version 2 of the License, or (at your option)
 * any later version.
 *
 * swt-vl is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St, Fifth Floor, Boston, MA 02110, USA
 * ---------------------------------------------------------------------------
 * $Id: Stack.java 1209 2012-04-10 08:41:55Z br $
 * ---------------------------------------------------------------------------
 */

package stack;

import java.util.EmptyStackException;
import java.util.Vector;

/**
 * Die Klasse Stack ist ein Stapel von Elementen des Typs T.<br/>
 * Ein Stapel ist eine zustandsorientierte Datenstruktur mit dem Prinzip LIFO
 * (last-in-first-out). Der Stapel hat n Elemente, n >= 0.
 *
 * Ein neu konstruierter Stack ist am Anfang leer, d.h. n == 0.
 *
 * @param <T> Typ der Elemente auf dem Stack
 */
public class Stack<T> {

  private Vector<T> content = new Vector<T>();
```

```
/**
 * push legt ein Element des Typs T auf den Stapel.
 * @param    item, ein Element vom Typ T
 * @pre      -
 * @post     item liegt auf dem Stapel, n' == n+1
 *           bisherige Elemente unverändert
 * @modifies this
 *
 */
public void push( T item ) {
  content.add( item );
}

/**
 * pop gibt das oberste Element des Stapels zurück und entfernt es vom Stapel.
 * @return   oberstes Element vom Stapel (vom Typ T)
 * @pre      n > 0, d.h. !this.isEmpty().<br/>
 * @post     n' == n-1 && restliche Elemente unverändert
 * @modifies this
 * @throws   EmptyStackException, falls Vorbedingung verletzt ist
 */
public T pop () {
  if ( content.isEmpty() ) {
    throw new EmptyStackException();
  }
  return content.remove( content.size()-1 );
}

/**
 * top gibt das oberste Element des Stapels zurück, ohne ihn zu verändern.
 * @return   oberstes Element vom Stapel
 * @pre      n > 0, d.h. !this.isEmpty().<br />
 * @post     alle Elemente unverändert, insbesondere n' == n
 * @throws   EmptyStackException, falls Vorbedingung verletzt ist
 */
public T top() {
  if ( content.isEmpty() ) {
    throw new EmptyStackException();
  }
  return content.elementAt( content.size()-1 );
}

/**
 * isEmpty gibt an, ob der Stapel leer ist.
 * @return    n == 0?
 * @post      Stapel ist unverändert
 */
public boolean isEmpty() {
  return content.isEmpty();
```

```
  }

}

/** ---------------------------------------------------------------------------
 * Test für Stack
 *
 * Copyright (c) 2010 - 2012
 * by Burkhardt Renz Technische Hochschule Mittelhessen.
 *
 * swt-vl is free software; you can redistribute it and/or modify it under
 * the terms of the GNU General Public License as published by the Free
 * Software Foundation; either version 2 of the License, or (at your option)
 * any later version.
 *
 * swt-vl is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for
 * more details.
 *
 * You should have received a copy of the GNU General Public License along
 * with this program; if not, write to the Free Software Foundation, Inc.,
 * 51 Franklin St, Fifth Floor, Boston, MA 02110, USA
 * ---------------------------------------------------------------------------
 * $Id: StackTest.java 1177 2012-03-21 13:28:33Z br $
 * ---------------------------------------------------------------------------
 */
package stack;

import java.util.EmptyStackException;

import org.junit.Test;
import static org.junit.Assert.*;


public class StackTest {

  @Test
  public void IntStackUsage() {
    Stack<Integer> stp = new Stack<Integer>();
    assertTrue( stp.isEmpty() );

    stp.push(2);
    stp.push(7);
    stp.push(1);
    stp.push(8);

    assertEquals( (Integer)8, stp.top() );

    stp.push(2);
```

```
    assertEquals( (Integer)2, stp.pop() );
    assertEquals( (Integer)8, stp.top() );
  }

  @Test(expected = EmptyStackException.class)
  public void IntStackMisuse1() {
    Stack<Integer> stp = new Stack<Integer>();
    stp.top();
  }

  @Test(expected = EmptyStackException.class)
  public void IntStackMisuse2() {
    Stack<Integer> stp = new Stack<Integer>();
    @SuppressWarnings("unused")
    int item = stp.pop();

  }
}
```

Lektüre:

MIT Department Electrical Engineering and Computer Science
Kurs 6,170 Laboratory in Software Engineering, Fall 2005
Lecture Notes L4 Specifications
http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-170-laboratory-in-software-engineering-fall-2005/lecture-notes/lec4.pdf

sowie die Java-Dokumentation der Klasse Stack<E>
http://docs.oracle.com/javase/7/docs/api/java/util/Stack.html

Burkhardt Renz
Technische Hochschule Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 3.1 – 10. April 2012