II. Vom Modell zum Programm
1. Klassen definieren

- Attribute
  - Typ
  - public, private oder protected
  - static ?
- Methoden
  - Signatur
  - static ?
- Vererbung
  - public oder private
  - mehrfach ?
- Konstruktoren und Destruktoren

Verhaltensregeln:

- Vererbungsstruktur übernehmen
- Innerer Details und Datenstrukturen verstecken
  $\Rightarrow$ Klasse bleibt austauschbar
- Keine Variablen nach aussen reichen

```
public int hour() { return internal_hour; }
private int internal_hour;
private int internal_minute;

public int hour() { return internal_minute/60; }
private int internal_minute;
```

2. Verbindungen realisieren

z.B. über Zeiger („Java hat nur Zeiger“)

- von einer Seite

- von beiden Seiten

Verbindungen zu mehreren Objekten durch Container realisieren:

- verkettete Listen (Einfügen und Löschen)
- Felder (Zugriff über Index)
- Bäume
- Hashtables
- etc.

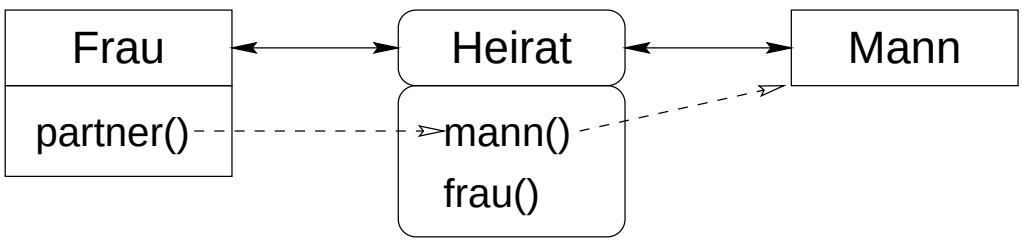
Alternative:

Container als Verzeichnis verwenden

Dorothea	E213
Uli	E215
Dagmar	E201
Annegret	E216
Sabine	E216
Thomas	E201

Verhaltensregeln:

- nicht über mehr als eine Verbindung gehen



III. Entwicklung großer Programme

1. Erst denken – dann Programmieren

Die Modellierung steht vor der Realisierung ✓

Namen aus der Modellierung beim Programmieren wiederverwenden

2. Spezifikation erstellen und offenlegen

⇒ Text am Schluss

3. Klassen zu Modulen zusammensortieren

Geometrie	Linie, Ebene, Kugel
GUI	Fenster, Menü, Maus-Ereignis
Zeichnen	Projektion, Ausgabe
Ein-/Ausgabe	VDA, STL

4. Programmier-Richtlinien festlegen

- Namenskonvention

```
SetValue();
setvalue()
set_value();
value_set();
```

- Parameterkonvention

```
output = input.FindMST();
input.FindMST(output);
```

- Dokumentation
- bis hin zu Art der Einrückung

Text hierarchisch einrücken

Gefahr: Man vergisst leicht die Blockklammern, wenn man eine Zeile durch weitere Befehle erweitert:

```
for (int Count=0; Count<MaxCount; Count++)
   Length[Count] = 0;
   Width[Count]  = 0;
```

Hilfe: Blockklammern immer setzen:

```
for (int Count=0; Count<MaxCount; Count++)
   { Length[Count] = 0; }
```

IV. KISS – **K**eep **I**t **S**imple and **S**tupid

```
function MM_swapImage() { //v2.0
var i,j=0,objStr,obj,swapArray=new Array,
oldArray=document.MM_swapImgData;
for (i=0; i < (MM_swapImage.arguments.length-2); i+=3) {
  objStr = MM_swapImage.arguments[(navigator.appName==
   'Netscape')?i:i+1];
  if ((objStr.indexOf('document.layers[')==0
  &&document.layers==null) ||
  (objStr.indexOf('document.all[') ==0&&document.all ==null))
objStr = 'document'+objStr.substring(objStr.lastIndexOf('.'),
objStr.length);
obj = eval(objStr); if (obj != null) {swapArray[j++] = obj;
 swapArray[j++] = (oldArray==null ||
 oldArray[j-1]!=obj)?obj.src:oldArray[j];
obj.src = MM_swapImage.arguments[i+2];
} }
document.MM_swapImgData = swapArray;
}
```

Grundprinzipien von KISS:

1. Nicht zu genial sein
2. Zwischenresultate explizit machen
3. Logische Einheiten separieren
4. Funktionen klein halten

1. Nicht zu genial sein

- Es ist nicht das Ziel, möglichst kurzen Quelltext zu schreiben.

- Möglichst keine Seiteneffekte benutzen.

- Gefährliche Befehle sind:

  - `a++` und `++a`

  - `a ? b : c`

  - `break` und `continue`

  Besonders dann, wenn man sie verschachtelt.

- Falls man „geniale“ Konstruktionen verwendet, sehr gut dokumentieren.

2. Zwischenresultate explizit machen

```
Ergebnis = foo(x*x+y*y, 0.5*bar(x,y,z), dx);
```

Was bedeuten die Werte?

```
double SkalarProdukt = x*x+y*y;
double Abstand = 0.5*bar(x,y,z);
Ergebnis = foo(SkalarProdukt, Abstand, dx);
```

3. Logische Einheiten separieren

```
void Hash_addline(struct Hash *graph,
                  StationNmbType from,
                  StationNmbType number,
                  StationNmbType to)
{
  struct HashValueType *node = Hash_find(graph, number);
  struct ItemCLine *lineit = NULL;

  /* sort nodes */
  if (from>to) { swapN(&from,&to); }

  /* look whether pair exists already */
  for (lineit=node->value->lines; lineit!=NULL; lineit=lineit->next)
    if (lineit->value.n1 == from && lineit->value.n2 == to)
      return;

  /* if not, add it */
  AddToList(CLine,node->value->lines);
  node->value->lines->value.n1 = from;
  node->value->lines->value.n2 = to;

}/* end of Hash_addline */

/************************************************************/

struct HashValueType* Hash_find(struct Hash* hash, HashKeyType k)
{
  CounterType pos = Hash_locate(hash, k);
  if (hash->data[pos].used)
    return hash->data + pos;
  else
    return NULL;
}
```

4. Funktionen klein halten

- Faustregel: nicht mehr als 10-15 Zeilen (plus Kommentar)
- Nicht mehrere verschiedene Dinge in einer Funktion (z.B. Initialisierung und Algorithmus)
- Arbeit und Kontrolle trennen

```
initialisierung(...)
algo1(...)
algo2(...)
```

V. Dokumentation

1. Dokumentation im Funktionskopf

- im Klartext beschreiben, was die Funktion macht

```
int faculty(int arg);
/* This function calculates the faculty. */
```

- den Sinn der Parameter beschreiben

```
int faculty(int arg);
/* This function calculates the faculty of <arg>. */
```

- notwendige Bedingungen an Parameter angeben

```
graph::findMST();
/*
   This function calculates a MST.
   The result is provided by the flag
   "treeEdge" of the edges.
   The result is undefined, if the graph is not
   connected.
*/
```

- insbesondere unberücksichtigte Ausnahmefälle explizit erwähnen.
- auch die Rückgabe dementsprechend beschreiben

Die Funktion soll benutzbar sein, ohne dass man sich den Quelltext ansehen muss

– auch für andere!

2. Dokumentation in der Funktion

- keine überflüssigen Kommentare

```
i++ // hier wird i um eins erhoeht
```

- sondern denn Sinn beschreiben

```
if (node.edge().number()==0)
  { // Fall 1: keine inzidenten Kanten
    ...
```

- Tricks dokumentieren

3. Gute Bezeichner verwenden

- möglichst den Sinn vollständig wiedergeben

- keine kryptischen Abkürzungen

```
double FGfacMin = 3.0;
```

- nicht denselben Bezeichner für unterschiedliche Dinge

```
Matrix::Invert();
Figure::Invert();
```

VI. Robusten Code schreiben

- Ein Programm sollte nie unkontrolliert aussteigen
  – auch nicht bei Fehleingabe
- Parameter in Funktionen überprüfen

```
Item Container::returnItem(int Index)
 {
   if (Index<0)
      ...
 }
```

- Invarianten herausarbeiten und überprüfbar machen

```
...
  algostep1();
  if (!graphConnected())
      { error("Graph disconnected after algostep1") };
  algostep2();
...
```

- Ausgabe des Programm-Zustandes planen
  $\Rightarrow$ leichteres Tracing

- Warnungen des Compilers ernst nehmen (und vermeiden)

- Andere Programme wie z.B. purify verwenden (und deren Fehlermeldungen ernst nehmen)

- Erst zum Schluss optimieren

- Dabei einen Profiler benutzen