Objektiv Denken II: Datenkapselung, Zugriffsmethoden und Vererbung in MATLAB

Ernesto Rico-Schmidt

Kurzfassung— **Im ersten Teil dieser Serie wurde als Einstieg in die objektorientierten Programmierung in Matlab ein einfaches Beispiel besprochen. Die Realisierung einfacher Konzepte der in Matlab wie das Klassenverzeichnis, der Konstruktor, die Umwandlungsmethoden und das Überladen von Methoden (Funktionen) und Operatoren wurden Anhand eines einfachen Beispiels erläutert.**

In diesem zweiten Teil werden die Realisierung der Datenkapselung, der Zugriffsmethoden (`get` und `set`), sowie der einfachen Vererbung besprochen. Als Grundlage dient ein Beispiel aus der Konstruktionsübung "Objektorientierte Programmierung" von Dipl.-Ing. Ausserhofer am Institut für Informationsverarbeitung und Computergestützte Neue Medien.[1]

I. Datenkapselung und einfache Vererbung in Matlab

UNTER Datenkapselung versteht man das Verstecken der Details der Implementierung einer Klasse bzw. eines Objekts. Die Kapselung trennt so die Schnittstelle einer Abstraktion von ihrer Implementierung.[2]

Man braucht sich dann nicht mehr darum zu kümmern, wie etwas implementiert wurde. Man verwendet lediglich die zur Verfügung gestellte Schnittstelle (Methoden). Der Zugriff auf die Attribute eines Objekts geschieht dann ebenfalls über eine vordefinierte Schnittstellen: die Methoden `get` und `set`.

Die Vererbung ist das zentrale Konzept der objektorientierten Programmierung bzw. Modellierung. Eine Hierarchie von Klassen wird aufgebaut, wo die die sogenannte Basisklasse alle gemeinsame Attribute enthält. Andere Klassen werden von dieser *abgeleitet*, die Attribute werden vererbt und die Funktionalität (Methoden) kann erweitert oder angepaßt werden.

In diesem Artikel wird ein Wasserversorgungssystem, bestehend aus Quellen, (source) Senken (sink), Tanks (tank), Ventile (valve), und Leitungen (pipe) modelliert. Diese Komponenten werden von der Klasse component abgeleitet.

II. Die Klasse component

Die Klasse component ist die Basisklasse und enthält die Attribute Namen, Ab- und Zufluß, sowie Druck und Gegendruck in Abfluß- und Zuflußrichtung. Diese beschreiben und charakterisieren den Zustand einer Komponente.

Alle abgeleiteten Klassen spezialisieren die Komponente, indem sie die Klasse mit zusätzlichen Attributen versehen und/oder ein anderes Verhalten (in Form von Methoden) einzuprägen.

[1]Näheres zur Vorlesung und Konstruktionsübung findet man im Web unter:
https://online.tugraz.at/tug_online/lv.detail?corg=2367&clvnr=62354,
https://online.tugraz.at/tug_online/lv.detail?corg=2367&clvnr=60838

A. Der Konstruktor

Der Konstruktor ermöglicht drei Aufrufe, die Unterscheidung zwischen den Aufrufe erfolgt über die Anzahl der Eingangsparameter, `nargin`.

Kein Argument Erhält der Konstruktor kein Argument, dann wird ein *default*-Objekt erzeugt. Das Objekt wird dann instantiiert, indem `class` aufgerufen wird. Das Objekt wird anschließend zurückgegeben.

Objekt als Argument Ist das Argument im Aufruf ein Objekt derselben Klasse (component), dann wird einfach eine Kopie des Objekts zurückgegeben. Ob es sich um ein Objekt derselben Klasse handelt, wird mit der Funktion `isa` überprüft.

Daten als Argumente Werden Argumente übergeben, die kein Objekt derselben Klasse sind, dann wird Anhand dieser Daten (Parameter) ein neues Objekt erzeugt, mit einem `class`-Aufruf instantiiert und anschließend wieder zurückgegeben.

`@component/component.m`

```
function c = component(varargin)
% COMPONENT class constructor

switch nargin
 case 0         % default object
  c.name = 'none';
  c.inflow = 0;
  c.outflow = 0;
  c.inflowPress = 0;
  c.outflowPress = 0;
  c.inflowRes = 0;
  c.outflowRes = 0;
  c = class(c, 'component');
 case 1         % copy object
  if (isa(varargin{1}, 'component')),
    c = varargin{1};
  else
    error('wrong input argument type');
  end
 case 7         % new object
  c.name = varargin{1};
  c.inflow = varargin{2};
  c.outflow = varargin{3};
  c.inflowPress = varargin{4};
  c.outflowPress = varargin{5};
  c.inflowRes = varargin{6};
  c.outflowRes = varargin{7};
  c = class(c, 'component');
 otherwise
  error('wrong number of input arguments');
end
```

B. Die `get`- und `set`-Methoden

Ein Zugriff auf die Felder des Objekts außerhalb der Klasse (d.h. außerhalb des Klassenverzeichnis) ist nicht möglich. Um auf die Attribute eines Objekts zugreifen zu

können, müssen die Methoden `get` und `set` implementiert werde, mit denen dann eine Schnittstelle festgelet wird, denn man kann den Zugriff auf die Attribute einschränken, wenn es nötig ist bzw. bei abgeleiteten Klassen erweitern.

Die Methode `get` gibt den Wert eines Attributs zurück. Diese Funktionalität wird in den abgeleiteten Klassen verwendet, um auf den Basis-Teil des Objekts zuzugreifen. Siehe die `get`-Methode der Klasse tank.

@component/get.m

```
function val = get(c, attr)
% GET method for COMPONENT class

switch attr
 case 'name'
  val = c.name;
 case 'inflow'
  val = c.inflow;
 case 'outflow'
  val = c.outflow;
 case 'inflowPress'
  val = c.inflowPress;
 case 'outflowPress'
  val = c.outflowPress;
 case 'inflowRes'
  val = c.inflowRes;
 case 'outflowRes'
  val = c.outflowRes;
 otherwise
  error(['attribute ', attr, ' is not defined']);
end
```

Die Methode `set` verändert den Wert eines Attributs. Da MATLAB kein *call by reference* kennt, und immer nur mit Kopien eines Objekts arbeitet, muß die Methode das veränderte Objekt zurückgeben und man muß ihn neu zuweisen:

```
>> c = set(c, 'inflow', 100)
```

Wiederum wird bei abgeleiteten Klassen auf diese Methoden zurückgegriefen, wenn man den Basis-Teil des Objekts verändern will. Siehe die `set`-Methode der Klasse tank.

@component/set.m

```
function c = set(c, attr, val)
% SET method for COMPONENT class

switch attr
 case 'name'
  c.name = val;
 case 'inflow'
  c.inflow = val;
 case 'outflow'
  c.outflow = val;
 case 'inflowPress'
  c.inflowPress = val;
 case 'outflowPress'
  c.outflowPress = val;
 case 'inflowRes'
  c.inflowRes = val;
 case 'outflowRes'
  c.outflowRes = val;
 otherwise
  error(['attribute ', attr, ' is not defined']);
end
```

C. Die *display*-Methode

Die Funktionalität der `display`-Methode wurde bereits in ersten Teil dieser Serie erläutert. Hier wird die `get`-Methode verwendet, um den Wert der Attribute abzufragen.

@component/display.m

```
function display(c)
% DISPLAY method for COMPONENT class

disp(['           ']);
disp(['     name: ' get(c, 'name')]);
disp(['   inflow: ' num2str(get(c, 'inflow')) ' l']);
disp(['  outflow: ' num2str(get(c, 'outflow')) ' l']);
```

III. Die Klasse tank

Aus den vorhandenen Komponenten des Systems wird hier die Klasse tank herausgegrifenn, denn sie zeigt einige Interessante Konzepte.

A. Der Konstruktor

Der Konstruktor ermöglicht wiederum drei Aufrufe, die anhand der Anzahl der Eingangsparameter unterschieden werden. Die Vorgangsweise um die Vererbung zu realisieren ist hier sichtbar.

1. Ein Objekt der Basis-Klasse component (`c`) muß zunächst instantiiert werden, indem der Konstruktor aufgerufen wird.
2. Die zusätzlichen Attribute des abgeleiteten Objekts (`t`) müssen initialisiert werden.[2]
3. `t` muß als Objekt der abgeleiteten Klasse tank mit `c` als Basis-Objekt gekennzeichnet werden. (`class(t, 'tank', c)`.)
   Der Basis-Teil des Objekts (`c`) ist dann im Feld `c.component` des neuen Objekts gespeichert. Siehe `get`- und `set`-Methoden.

[2] Enthält die abgeleitete Klasse keine zusätzliche Attribute, so muß das neue Objekt zumindest ein *dummy*-Attribut enthalten, siehe z.B. die Klassen source oder sink.

@tank/tank.m

```
function t = tank(varargin)
% TANK class constructor

switch nargin
 case 0         % default object
  c = component;
  t.maxOutflow = 0;
  t.level = 0;
  t.minLevel = 0;
  t.maxLevel = 0;
  t.volume = 0;
  t = class(t, 'tank', c);
 case 1         % copy object
  if (isa(varargin{1}, 'tank'))
    s = varargin{1};
  else
    error('wrong argument type');
  end
 case 5         % new object
  c = component(varargin{1}, 0, 0, 0, 0, 0, 0);
  t.maxOutflow = varargin{2};
  t.minLevel = varargin{3};
  t.maxLevel = varargin{4};
  t.volume = varargin{5};
  t.level = 0;
  t = class(t, 'tank', c);
 otherwise
  error('wrong number of input arguments');
end
```

B. Die *get*- und *set*-Methoden

Die Funktionalität der get-Methode wird hier erweitert, um auf die zusätzlichen Attribute zugreifen zu können. Um auf die Attribute des Basis-Objekts zuzugreifen, wird die get-Methode der Basis-Klasse verwendet. `t.component` ist ein Objekt der Klasse component.

@tank/get.m

```
function val = get(t, attr)
% GET method for TANK class

switch attr
 case 'name'
  val = get(t.component, 'name');
 case 'inflow'
  val = get(t.component, 'inflow');
 case 'outflow'
  val = get(t.component, 'outflow');
 case 'inflowPress'
  val = get(t.component, 'inflowPress');
 case 'outflowPress'
  val = get(t.component, 'outflowPress');
 case 'inflowRes'
  val = get(t.component, 'inflowRes');
 case 'outflowRes'
  val = get(t.component, 'outflowRes');
 case 'maxOutflow'
  val = t.maxOutflow;
 case 'minLevel'
  val = t.minLevel;
 case 'maxLevel'
  val = t.maxLevel;
 case 'volume'
  val = t.volume;
 case 'level'
  val = t.level;
 otherwise
  error(['attribute ', attr, ' is not defined']);
end
```

Die Funktionalität der set-Methode wird hier wiederum erweitert, um auf die zusätzlichen Attribute zugreifen zu können. Dabei ist es möglich Einschränkungen einzuführen. Hier dürfen z.B. Attribute wie `maxOutflow`, `maxLevel`, `minLevel` und `volume` nach dem Instantiieren nicht mehr verändert werden.

@tank/set.m

```
function t = set(t, attr, val)
% SET method for TANK class

switch attr
 case 'name'
  t.component = set(t.component, 'name', val);
 case 'inflow'
  t.component = set(t.component, 'inflow', val);
 case 'outflow'
  t.component = set(t.component, 'outflow', val);
 case 'inflowPress'
  t.component = set(t.component, 'inflowPress', val);
 case 'outflowPress'
  t.component = set(t.component, 'outflowPress', val);
 case 'inflowRes'
  t.component = set(t.component, 'inflowRes', val);
 case 'outflowRes'
  t.component = set(t.component, 'outflowRes', val);
 case 'maxOutflow'
  error(['attribute ', attr, ' can''t be changed']);
 case 'minLevel'
  error(['attribute ', attr, ' can''t be changed']);
 case 'maxLevel'
  error(['attribute ', attr, ' can''t be changed']);
 case 'volume'
  error(['attribute ', attr, ' can''t be changed']);
 case 'level'
  t.level = val;
 otherwise
  error(['attribute ', attr, ' is not defined']);
end
```

C. Die *display*-Methode

Hier wird die display-Methode der Basis-Klasse angewendet um den Basis-Teil des Objekts anzuzeigen. Die zusätzlich eingeführten Attribute werden gesondert behandelt.

@tank/display.m

```
function display(t)
% DISPLAY method for TANK class.

display(t.component)
disp(['    level: ' num2str(get(t, 'level')) 'l']);
```

IV. Zusammenfassung

Die Realisierung der Datenkapselung und Zugriffsmethoden get und set, sowie der einfachen Vererbung in MATLAB wurden anhand eines Beispiels erläutert.

Alle M-Files zu dem Beispiel findet man im Web:
http://www.cis.tugraz.at/ieee/ik/Juni-2000/objektiv.html

In der nächsten Ausgabe werden das Beispiel und die Serie abgeschlossen. Die Beinhaltung wird behandelt um die Komponenten zu einem System zusammenfassen zu können, und dieses anschließend zu simulieren.

V. Eine Anmerkung

Bjarne Stroustrup, der Erfinder der Programmiersprache C++, weist in [1] auf den Unterschied hin, der zwischen dem Unterstützen (*support*) und dem Ermöglichen (*enable*) von objektorientierter Programmierung besteht.

Eine Programmiersprache *unterstützt* demnach das objektorientierte Paradigma, wenn es einfach, sicher und effizient ist, so zu programmieren. Wird dies lediglich durch die Sprache *ermöglicht*, dann ist es mühsam und trickreich objektorientierte Programme zu schreiben.

MATLAB ist *de facto* zum Standard geworden, wenn es um technische Anwendungen (wie Simulation, Datenanalyse und -Visualisierung) geht, und die objektorientierte Programmierung wird erst seit der Version 5 *unterstützt*. Es ist gewiß nicht die beste, oder natürlichste Umgebung um objektorientierte Programme zu schreiben, aber durchaus eine brauchbare.

Schrifttum

[1] Bjarne Stroustrup: *What is Object-Oriented Programming?* (1991 revised version). Proc. 1st European Software Festival. February, 1991.

[2] Grady Boochs: *Object-Oriented Analysis and Design with Applications* Second Edition, 1994