Objektorientierte Programmierung (OOP) mit Matlab - Ein Crashkurs

Sebastian Sahli

31. Mai 2016

Grundbegriffe (I)

- **Objektorientierung**: „Sichtweise auf komplexe Systeme, bei der ein System durch das Zusammenspiel kooperierender Objekte beschrieben wird“ [1]
- **Objekte**: Unscharf gefasster Begriff. Objekte werden charakterisiert durch:
  - Die Dinge, die sie *haben* (ihre Eigenschaften/ Attribute)
  - Die Dinge, die sie *können* (ihre Methoden)
- Objekte müssen nicht materiell sein

[1] `https://de.wikipedia.org/wiki/Objektorientierung`

Grundbegriffe (II)

- **Objektorientierte Programmierung (OOP)**:
  - Programmierstil, der die Objektorientierung umsetzt
  - Grundidee: Modellierung des Programms mithilfe von Objekten, deren Zusammenspiel das zu lösende Problem beschreibt
  - Vorstellung, dass alles aus Objekten aufgebaut ist bzw. sich alles als Objekt darstellen lässt
  - Insbesondere ist ein objektorientiertes Programm keine lineare Abfolge von Anweisungen

Klassen

- Wichtigstes Konzept der OOP: die **Klasse**
  - Beschreibt abstrakt eine Menge von Objekten mit gleichen Eigenschaften und Fähigkeiten
  - Sozusagen ein *Bauplan* für eine bestimmte Sorte/ einen bestimmten *Typ* von Objekten
  - In der Klasse werden die Attribute und Methoden des zu beschreibenden Typs angegeben

Klassen in Matlab

- In Matlab

```
classdef Name
  properties
    //Attribute kommen hierhin
  end

  methods
    //Methoden kommen hierhin
  end
end
```

Beispiel

- Beispiel **Polynome** (in $\mathbb{R}$):
  - Ein Polynom besitzt *Koeffizienten* (Eingenschaft)
  - Ein Polynom kann an einer Stelle $x \in \mathbb{R}$ *ausgewertet* werden (Methode)

Beispiel (in Matlab)

```
classdef Name
  properties
    coefficients
  end

  methods
    function y = evaluate(obj, x)
      y = ... ;
    end
  end
end
```

Klassen in Matlab (II)

- Attribute werden einfach aufgelistest
- Methoden können immer nur auf konkrete *Objekte* der Klasse angewendet werden (was soll die Auswertung eines abstrakten Polynoms an der Stelle 5 sein?)
- Aus diesem Grund ist der erste Parameter (hier `obj` in der `evaluate`-Methode) einer Klasenmethode *immer* ein konkretes Objekt der entsprechenden Klasse
- Es ergibt sich eine Frage: Wie erzeugt man ein konkretes Objekt einer Klasse?

Konstruktoren

- Antwort: **Konstruktor**
  - Spezielle Methode, die den gleichen Namen wie die Klasse hat
  - Rückgabewert ist ein Objekt der Klasse
  - Im Konstruktor werden die Attribute des Klassenobjekts auf konkrete Werte gesetzt
  - Hat als ersten Parameter *kein* Objekt der Klasse (ein solches soll ja erst erzeugt werden)
  - ~~Eine Klasse kann durchaus mehrere Konstruktoren haben~~ Matlab says No

Konstruktoren in Matlab (I)

```
classdef Polynom

  properties
    //...
  end

  methods

    //Konstruktor
    function obj = Polynom(coeff)
      obj.coefficients = coeff;
    end

  end
end
```

- Attribute werden wie bei `struct`-Variablen gesetzt

Konstruktoren in Matlab (II)

- Wird kein Konstruktor angegeben, so wird implizit immer ein parameterloser Konstruktor erzeugt, der gar nichts tut
- Aufruf des Konstruktors:

```
f = Polynom([1 0 3]);
```

- Das Objekt `f` ist dann vom *Typ* `Polynom` bzw. eine *Instanz* der Klasse `Polynom`
- Für das konkrete Objekt `f` kann dann die `evaluate`-Methode aufgerufen werden
- Zwei syntaktische Möglichkeiten:
  - `evaluate(f, 5)` oder
  - `f.evaluate(5)`

Typabfrage

- Man kann mittels der Methode `isa` abfragen, ob ein Objekt Instanz einer gegebenen Klasse ist
- Syntax:

```
b = isa(f, 'Polynom');
```

- Achtung: Matlab kennt *kein* Typsystem, d.h.:
  - Prinzipiell kann einer Funktion immer ein Objekt einer beliebigen Klasse übergeben werden
  - Erst wenn versucht wird, auf ein nicht-existentes Attribut/ Methode zuzugreifen, wirft die Laufzeitumgebung einen Fehler
  - Überprüfung, ob das Argument vom richtigen Typ ist, muss manuell geschehen

Sichtbarkeiten (I)

- Oft sollen Daten nachträglich von außen nicht mehr geändert werden können
- Ebenso sollen interne Hilfsfunktionen dem Benutzer verborgen bleiben
- Dazu dienen **Sichtbarkeitsmodifizierer**

Sichtbarkeiten (II)

- Zu Beginn eines `properties`- oder `methods`-Blocks kann geregelt werden, wer darauf zugreifen darf. Mögliche Werte sind
  - `public`: Jeder darf auf die Methoden/ die Atrribute in diesem Block zugreifen
  - `protected`: Siehe später
  - `private`: Auf die Methoden/ Attribute in diesem Block kann nur innerhalb der Klasse zugeggriffen werden
- Jede Klasse kann mehrere solcher Blöcke mit unterschiedlichen Sichtbarkeiten enthalten

Sichtbarkeiten (III)

- Syntaktisches Beispiel:

```
classdef bla

  properties(Access = private)
    x
    y
  end

  methods(Access = public)
    function y = doSomething(obj)
      y = 42;
    end
  end

  methods(Access = private)
    function helper(obj)
      //Hier passiert nix.
    end
  end
end
```

Sichtbarkeiten (IV)

- Im Beispiel ist dann

```
T = bla;
T.x
```

  nicht mehr erlaubt
- Bei Attributen können der Lese- und der Schreibzugriff mittels `GetAccess` und `SetAccess` getrennt eingestellt werden
- Beispiel:

```
properties(GetAccess = public, SetAccess = private)
  x
  y
end
```

- Wird eine Sichtbarkeit nicht explizit angegeben, so wird sie immer als `public` angenommen

Vererbung (I)

- Zweites wichtiges Konzept der OOP: **Vererbung**
- Erlaubt es
  - bestehende Klassen zu erweitern
  - die Implementierung einzelne Methoden einer Klasse zu ändern (sog. *Überschreiben* von Methoden)
- Vorteil: Keine Änderungen an bestehendem Code
- Wird eingesetzt, um Situationen zu modellieren, in denen eine Klasse von Objekten *spezieller* ist als eine andere

Vererbung (II)

- Mathematisch angehauchtes Beispiel:
  - Eine Klasse `Function` beschreibt ganz allgemein das Konzept einer Funktion (auf $\mathbb{R}$) dar und stellt eine Methode `evaluate` zur Auswertung an einer konkreten Stelle zur Verfügung
  - Die Klasse `Polynom` *spezialisiert* `Function` (da jedes Polynom insbesondere eine Funktion ist) und erweitert diese um das Attribut `coefficients` (die Koeffizienten sind Polynom-spezifisch), und ändert die Implementierung der `evaluate`-Methode

Vererbung in Matlab (I)

- Vererbung geschieht durch Angabe der *Oberklasse* (d.h. der Klasse, von der geerbt werden soll) bei der Klassendefinition:

```
classdef Klasse < Oberklasse
  //...
end
```

- Die Unterklasse hat dann sofort alle Methoden und Attribute der Oberklasse zur Verfügung

Vererbung in Matlab (II)

- Achtung: Besitzt eine Klasse keinen parameterlosen Konstruktor, so *muss* als erster Befehl im Konstruktor einer Unterklasse ein Konstruktor der Oberklasse aufgerufen werden:

```
classdef Klasse < Oberklasse

  methods

    //Konstruktor
    function obj = Klasse(args)
      obj@Oberklasse(Argumente, des,
          Oberklassenkonstruktors);

      //...
    end
  end
end
```

- Stellt sicher, dass alle Atrribute, die von der Oberklasse bereitgestellt werden, korrekt initialisiert werden

Vererbung in Matlab (Ergänzung zu II)

- Stimmt nicht ganz!
- Nur die `obj`-Variable darf vor dem Aufruf des Superklassen-Konstruktors nicht verwendet werden!
- Lokale Variablendefinitionen sind möglich
- Beispiel

```
function obj = Klasse(args)
  x = 42;
  y = 5;

  obj@Superklasse(x,y);
end
```

Vererbung in Matlab (III)

- Neue Attribute/ Methoden werden durch Definition neuer `properties`- und `methods`-Blöcke hinzugefügt
- Bereits vorhandene Methoden werden durch einfache Neudefinition überschrieben

Vererbung (III)

- Spannende Frage: **Was bringt's?**
  - Erweiterung von Code möglich, ohne bisherigen Code ändern zu müssen
  - An Stellen, an denen ein Objekt einer Klasse erwartet wird, ist immer auch ein Objekt einer Unterklasse möglich
  - Erbt `Klasse` von `Oberklasse` und ist `X` vom Typ `Klasse`, so ist `isa(X, 'Oberklasse')` wahr!
  - Ermöglicht es, eine Unterklasse zu erstellen, die gewisse Dinge anders implementiert und diese anstatt der Oberklasse zu verwenden
  - Der umliegende Code, der die Klasse verwendet, muss dafür **nicht** geändert werden

Abstrakte Klassen (I)

- Manchmal ist die Implementierung einer Methode nicht von vornherein klar, nur, *dass* es eine solche geben muss
- Beispiel `Function`-Klasse von oben: Die `evaluate`-Methode *muss* vorhanden sein, ihre konkrete Implementierung ist aber von der konkreten Art der Funktion abhängig
- Ein Polynom, das über seine Koeefizienten definiert ist, muss anders ausgewertet werden als eine Funktion, die über ein function handle gegeben ist

Abstrakte Klassen (II)

- Lösung: **Abstrakte Methoden**
  - Methoden ohne Implementierung
  - Stehen in einem speziellen Block

```
methods(Abstract)
  //...
end
```

  - Werden ohne das Schlüsselwort `function`, ohne Rückgabewert und ohne Rumpf definiert:

```
evaluate(obj, x)
```

  - Von Klassen mit abstrakten Methoden können keine Objekte erzeugt werden, erst von einer Unterklasse, die die abstrakten Methoden überschreibt

Abstrakte Klassen (III)

Und der **Sinn**?

- Abstrakte Klassen stellen eine Art *Vertrag*, was gewisse Objekte können müssen
- Verschiebt die Implementierung auf später
- Erlaubt es, Objekte mit prinzipiell gleichen Fähigkeiten aber unterschiedlicher Implementierung unter einer Oberklasse zusammenzufassen
- In Funktionen kann ein Objekt der abstrakten Klasse als Argument erwartet werden, übergeben werden später Objekte konkret implementierter Kindklassen
- Ermöglicht abstraktere bzw. allgemeinere Formulierungen von Methoden

Handle-Klassen (I)

- Standarmäßiges Verhalten Verhalten von Klassenobjekten: Wird ein solches an eine Funktion übergeben, so wird eine Kopie angelegt
- Wird innerhalb der Funktion ein Attribut geändert, so betrifft diese Änderung nur die Kopie – das ursprüngliche Objekt wird *nicht* geändert
- Beispiel:

Handle-Klassen (II)

```
classdef bla
  properties
    x
  end

  methods
    function obj = bla(val)
      obj.x = val;
    end
  end
end
```

```
function blubb(obj)
  obj.x = 42;
end
```

Handle-Klassen (III)

- Def Aufruf

```
X = bla(5);
blubb(X);
z = X.x
```

  liefert `z = 5`!

- Abhilfe: Ableiten von der Klasse **handle**:
  - Handle-Klassen werden nur einmal im Speicher angelegt
  - Die Objektvariable ist lediglich ein *Zeiger* auf das Objekt im Speicher
  - Arbeitet eine Funktion auf einem Handle-Objekt, so arbeitet sie immer auf den Originaldaten – Änderungen nach außen werden also sichtbar!

Handle-Klassen (IV)

- Schreibt man also im Beispiel `classdef bla < handle`, so liefert der Aufruf

```
X = bla(5);
blubb(X);
z = X.x
```

  hier tatsächlich `z = 42`.

Operatorüberladung (I)

- Betrachte wieder die `Polynom`-Klasse von oben
- Sind `f = Polynom([1 0 3])` und `g = Polynom([2 3 0 1])` Polynomobjekte, so wäre es wünschenswert, wenn auch

  ```
  h = f + g;
  ```

  ein Polynomobjekt liefert mit passenden Koeffizienten
- In Matlab ist das möglich

Operatorüberladung (II)

- Dazu implementiere in einer Klasse eine Methode mit einem Parameter

```
function y = mplus(obj, other)
  y = ... ;
end
```

- Der Aufruf `f.plus(g)` bzw. `plus(f, g)` lässt sich dann ersetzen durch `f+g`
- Analog ermöglicht eine Implementierung von `mtimes` einen Aufrif der Form `f * g`
- Analog für „/“, etc.

Operatorüberladung (III)

- Erlaubt prinzipiell auch das Anwenden von Operatoren auf Objekte unterschiedlicher Typen (z.B. $\lambda \cdot f$ für einen Skalar λ und eine Funktion f)
- Aber Achtung: Problematisch wird's, wenn `f + g` für Objekte von unterschiedlichen Klassen aufgerufen wird, die *beide* `mplus` implementieren – welches wird das verwendet??

Strukturierung des Quellcodes

- Klassen lassen sich in speziellen Ordnern (zum Beispiel inhaltlich) gruppieren
- Der Name eines solchen Ordners ist durch ein vorangestelltes `+` gekennzeichnet
- Ist `Klasse` eine Klasse im Ordner `+Ordner`, so lässt sie folgendermaßen ansprechen (wenn der Ordner im aktuellen Matlab-Ordner enthalten ist):

```
Ordner.Klasse
```

- Beispiel: `obj = Ordner.Klasse`
- Ein solcher Ordner bildet mit den in ihm enthaltenen Klassen ein *Package* bzw. einen *Namensraum*

Weiterführende Informationen

- Gute Anlaufstelle: http://de.mathworks.com/help/matlab/object-oriented-programming.html
- Wer wirklich *alles* wissen will: http://de.mathworks.com/help/pdf_doc/matlab/matlab_oop.pdf