Veranschaulichung des doppelhierarchischen Ansatzes von CYBOP durch die Programmiersprache Python

Max Brauer
<max.brauer@inqbus.de>

5. September 2012

Abstract

Cybernetics Oriented Programming (CYBOP) stellt den Versuch dar, Lösungen aus der menschlichen Denkens- und Verhaltensweise in die Programmierung einfließen zu lassen. Dieser Ansatz hat sich bereits in nahezu allen anderen Wissenschaftsbereichen durchgesetzt. Lediglich in der Programmierung ist der Mensch den Gegebenheiten der Maschienen unterworfen. Gemündet ist dieses Vorhaben in der XML-basierten Programmiersprache CYBOL und dem Interpreter CYBOI[1].

Die vorliegende Arbeit versucht den Ansatz der in CYBOP genutzten Doppelhierarchie[2, 7.3.3 Double Hierarchy, Seite 235] auf die Programmiersprache Python zu übertragen, um so die Vorteile dieses Vorgehens zu verdeutlichen und Entwicklern den Einstieg in CYBOL zu erleichtern.

Inhaltsverzeichnis

1 Einleitung

Schon seit vielen tausend Jahren beobachtet der Mensch die Natur und versucht das daraus resultierende Wissen auf den täglichen Gebrauch zu adaptieren. Ein Beispiel für

eine solche Adaption ist das Einsetzen des Lotuseffektes: Dank Nanopartikeln ist es möglich eine Oberfläche so zu gestalten, das keine Schmutzpartikel an ihr haften bleiben.[3]

Christian Heller beschreibt seine Motivation für das Buch „Cybernetics Oriented Programming“ im Kapitel 5 „Extended Motivation“ mit folgenden Worten:

> „Inspect solutions of various other disciplines of science, phenomenons of nature, and apply them to software engineering ... in order to find out if existing weaknesses can be eliminated.“[2, Seite 161]

Auch er versucht diesen Ansatz, welcher wieder und wieder zu Durchbrüchen in der Wissenschaft geführt hat: Durch das heranziehen von Naturphänomenen und Ergebnissen aus anderen Wissenschaftsdisziplinen sollen Schwächen in modernen Programmierparadigmen beseitigt werden.

Durch die Beschäftung mit dem Körper, dem Geist, dem menschlichen Denken und vielen anderen technischen und wissenschaftlichen, aber auch philosophischen Punkten hat sich der Ansatz einer »Doppelhierarchie« herauskristallisiert.

Das nachfolgende Kapitel beschäftigt sich mit der Herleitung von einem »Single Model«, hin zu einem »Hierarchischem Modell«, sowie dem zu Grunde liegenden Schema um zu verdeutlichen, wieso von einer »Doppelhierarchie« gesprochen wird. Am Ende des Kapitels, wird für dieses Wissen eine mögliche praktische Anwendung vorgestellt.

Das abschließende Kapitel stellt den doppelthierarchischen Ansatz mit Hilfe der Programmiersprache Python dar und versucht das zuvor gefundene Beispiel zu realisieren.

2 CYBOPS Doppelhierarchie

Eine essenzielle Erkenntnis der Forschungen von Christian Heller war die, dass alles eine gewisse Hierachie beinhaltet. Beispiele dafür bringt er im Kapitel 7.3.1 „Knowledge Ontology“: Bücherein enthalten Räume mit Büchern. Die Bücher enthalten Kapitel, welche Absätze enthalten. Diese wiederum enthalten Sätze, welche aus Wörtern bestehen[2, Seite 231]. Im selben Kapitel werden noch weitere Beispiele genannt.

Der doppelthierarchische Ansatz ist – in Kombination mit dem Schema – der zentrale Verständnispunkt der kybernetikorientierten Programmierung. Dabei werden zwei Hierarchien verwendet: Ein Wissensbaum, welcher die Beziehungen der Objekte untereinander hierarchisch abbildet, stellt die erste Hierarchie dar. Teilweise wird dieser Strang auch als „Mikro-/Makro-Kosmos“ bezeichnet. Die Metainformationen an den einzelnen Objekten die Zweite.

Dieses Kapitel beschäftigt sich mit der Herleitung der Hierarchie (Kapitel 2.1), der Herleitung des im Buch oft erwähnten »Schemas« (Kapitel 2.2), sowie des Zusammenschlusses zur Doppelhierarchie und deren praktischen Nutzen (Kapitel 2.3).

2.1 Herleitung des hierarchischen Modelles

Während der Erstellung von CYBOP ging die Entwicklung der Datenstruktur von einem „Single Model“ Ansatz über einen semistrukturierten Ansatz hin zu einem finalen Ansatz, in welchem die Struktur über eine Hierarchie abgebildet wurde.

Das »Single Model« ist die heutzutage am weitesten verbreitete Struktur. „Singel Model“ steht für „einziges Model“. Hierbei wird für jeden Typen eine eigene Klasse angelegt, welche anschließend untereinander

verweisen. Auf Seite 218 zeigt Heller[2] hierfür ein Beispiel: Die Klasse »Person« hat mehrere Attribute wie zum Beispiel das Geburtsdatum, Geschlecht, der Name oder die Adresse. Der Name und die Adresse sind dabei komplexere Strukturen und haben jeweils eine eigene Klasse. Im selben Kapitel werden noch einige Probleme an diesem Ansatz aufgezeigt. So ist zum Beispiel eine Erweiterung bestehender Strukturen nur schwer möglich.

Darauf aufbauend existiert das „Semi Structured Model“, was übersetzt „teilweise strukturiertes Modell“ bedeutet. Hierbei werden laut Heller[2, Seite219] »Named Values« verwendet. Diese werden in Listen gespeichert und können dynamisch hinzugefügt und verändert werden. Außerdem haben sie alle den einheitlichen Typ »Item«.

Das hierarchische Model baut nun wiederum auf das „Semi Structured Model“ auf. Heller beschreibt dieses Model im Kapitel 7.2 Design Reflections unter der Überschrift „Hierarchical Model“[2, Seite 220]. Es gibt lediglich eine verbleibende Klasse, welche keine fest definierten Attribute hat. Diese werden dynamisch hinzugefügt und verwenden dabei dasselbe Model. So entsteht eine Hierarchie, welche stark dem menschlichen Denken angelehnt ist.

Diese Struktur findet im cybernetischen Ansatz ihe Verwendung. Sie stellt die erste Hierarchie dar: den Mikro-/Makro-Kosmos. Alle Dinge, egal wie klein oder groß, sind Teil von größeren. Im Umkehrschluss gilt, alle Dinge enthalten kleinere. Nehmen wir hierzu folgendes Beispiel: Es existiert ein beliebiges Sonnensystem. Dieses existiert wiederum in unserer Sonnenstraße (Makro-Kosmos) und kann Planeten enthalten (Mikro-Kosmos). Dies könnte man nun weiter fortführen. Neben dem Mirko-/Makro-Kosmos gibt es noch eine zweite Hierarchie, weshalb von einer „Doppelhierachie“ gesprochen wird. Die zweite Hierarchie ist das Schema und wird im nachfolgenden Kapitel vorgestellt.

2.2 Schema

Der erste Teil der Doppelhierarchie wurde im voran gegangenen Kapitel erläutert. Nun fehlt lediglich noch der zweite Teil: Das Schema. Abbildung 1 ist dem Buch »Cybernetics Oriented Programming« entnommen und zeigt den Aufbau des Schemas, welchem ein jedes Objekt (Ausnahmen bieten hier primitive Datentypen) in CYBOP zu Grunde liegen. Es braucht einen eindeutigen Bezeichner (»name«) sowie eine Abstraktion (»abstraction«). Letzteres ist mit dem „Typ“ eines Objektes in anderen Programmiersprachen vergleichbar. Alle weiteren Informationen werden dynamisch in »details« gespeichert. Die in Kapitel 1 »1 zu n Relation« in Bezug auf »model« wird für die im Kapitel 2.1 vorgestellte Hierarchie benötigt. Diese Informationen stammen aus dem Kapitel 7.3.2 des Buches „Cybernetics Oriented Programming“. Für weitere Informationen empfehle ich dieses Kapitel, sowie die Kapitel 7.1.3 und 7.2.5.[2]

Nachdem nun kurz der Aufbau des Schemas vorgestellt wurde, wird sich im nächsten Kapitel mit der aus dem Schema sowie der hierarchischen Struktur entstehenden Doppelhierarchie beschäftigt. Außerdem wird auf einen praktischen Nutzen eingegangen.

2.3 Die Doppelhierarchie und deren praktischer Nutzen

In den Kapiteln 2.1 und 2.2 wurde das hierarchische Modell, sowie das in CYBOP verwendete Schema vorgestellt. Beide Modelle ergeben zusammen den doppelthierarchischen Ansatz, welche den gedanklichen Kern beim kybernetischen Programmieren darstellen.

Es gibt mehrere Objekte, welche in einer hierarchischen Struktur abgelegt werden. Dabei sollte die hierarchische Verknüpfung der Objekte logisch erfolgen um eine Nachvollziehbarkeit zu ermöglichen. Jedes

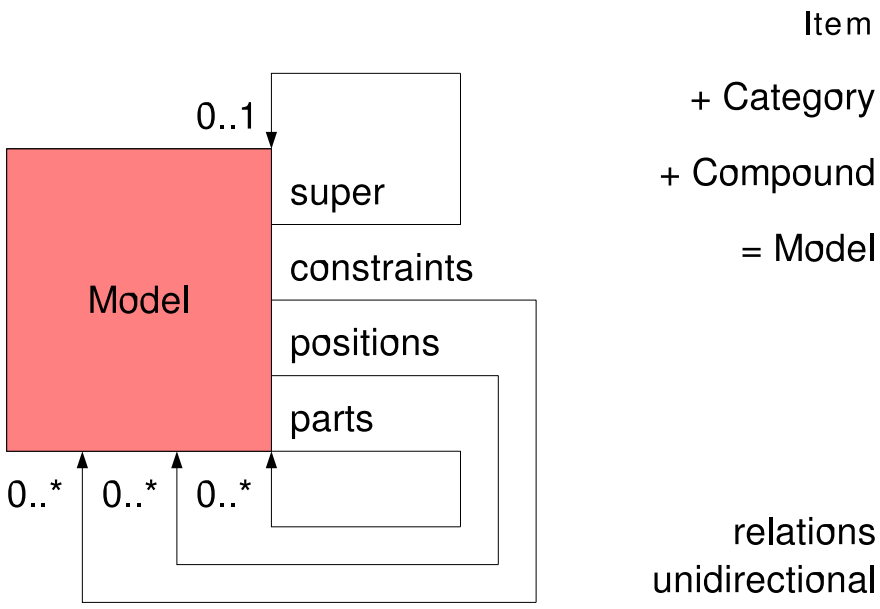

Abbildung 1: Knowledge Schema mit Metainformationen[2, Seite 234]

einzelnen Objekte verfügt über Metainformationen, welche dynamisch hinzugefügt und bearbeitet werden können. Diese Metainformationen können ebenfalls in einer hierachischen Form vorliegen.

CYBOP stellt nicht den erste Versuch dar, in der Informatik daten hierchisch zur Verfügung zu stellen, zu verwenden oder persistent abzulegen. So seien an dieser Stelle objektorientierte Datenbanken wie die »ZODB« für Python[4] oder »db40« für Java oder .NET[5] zu nennen, welche meist die Möglichkeit einer hierachischen Verwendung mitbringen. Der Vorteil solcher Datenbanken liegt in der fehlenden Abstraktionsschicht zwischen Datenbank und Anwendung. Bei relationalen Datenbanken ist eine »ORM« (Object Relational Mapper) Schicht nötig um die Daten aus der Datenbank in Objekte zu serialisieren und nach dem Ändern wieder zu deserialisieren und zu speichern. Allerdings werden die hierarchisch vorliegenden Objekte anschließend mit anderen Programmierparadigmen vermischt. Dadurch gehen große Teile der Verständlichkeit und des hierarchischen Ansatzes verloren.

Das nachfolgende Kapitel versucht genau das zu überbrücken: Es wird der doppelthierarchische Ansatz von CYBOP versucht in der Objektorientierten Programmiersprache Python zur Verfügung zu stellen. Zur persistenten Datenhaltung wird die zuvorgenannte ZODB[4] verwendet. Bei der ZODB handelt es sich um die „Zope Object Database“, einer objektorientierten Datenbank, welche dem Zope-Projekt[6] entsprungen ist.

3 Darstellung mittels Python

Der kybernetischorientierte Ansatz nach Heller, welcher im Kapitel 2 vorgestellt wurde, setzt einen hohen Grad an dynamischer Programmierung vorraus. Diesen hatte Heller in der Programmiersprache CYBOL[1] erschaffen. Aber auch andere

Programmiersprachen können diese Anforderung erfüllen.

Python ist eine sehr dynamische Programmiersprache, in welcher es mit wenig Aufwand möglich ist den doppelthierarchischen Ansatz zu implementieren. Dieses Kapitel beschäftigt sich mit diesem Vorhabden und erläutert, welche weiteren Möglichkeiten in Python über Meta-Programmierung möglich wären.

3.1 Der doppelthierarchische Ansatz in Python

Python ist eine dynamische Programmiersprache, welche viel Funktionalität mit sich bringt um den doppelthierarchischen Ansatz in Python zu nutzen. So verfügt jedes Objekt in Python über eine `__setattr__(prop, value)`-Methode. Ihr kann man einen Variablenbezeichner, sowie einen Wert übergeben und auf diesem Wege dynamisch Attribute hinzufügen. In folgendem Beispiel wird eine Klasse `Testobject` erstellt, welche nicht über das Attribut `name` verfügt. Dieses wird erst dynamisch hinzugefügt. Anschließend kann duch Punkt-Notation darauf zugegriffen werden:

```
>>> class Testobject(object):
...     pass
...
>>> myobject = Testobject()
>>> myobject.name
Traceback (most recent call last):
  File "<stdin>", line 1, in <
      module>
AttributeError: 'Testobject'
    object has no attribute 'name'
>>> myobject.__setattr__('name', '
    Testobject')
>>> myobject.name
'Testobject'
```

Zum Auslesen einmal gesetzter Attribute gibt es ebenfalls eine Funktion in Python: `__getattribute__(prop)`. Damit kann über den Namen des Attributes desen Wert zurückgegeben werden. Auf das `Testobject` Beispiel angewendet, sieht dies wie folgt aus:

```
>>> myobject.__getattribute__('
    name')
'Testobject'
```

Dieses Wissen genügt, um die notwendigen Funktionen in eine eigene Klasse zu implementieren um das dynamische hinzufügen und lesen von Attributen an Objekten zu ermöglichen. Funktionen, deren Namen in Python mit `__` beginnen, sind als private Funktionen zu betrachten. Daher werden diese Funktionen in `get_property()` beziehungsweise `add_property()` gekapselt. Um auch jederzeit auf alle verfügbaren Attribute zugreifen zu können, wird eine Liste zur Datenhaltung der Properties vewendet. Außerdem wird, gemäß nach den Anforderungen an das Schema, der `name` (eindeutiger Bezeichner) sowie der `path` (eindeutiger Pfad zum Objekt im Wissensbaum) gespeichert. Der Pfad ist notwendig um anhand des Objektes herausfinden zu können, in welchem Punkt des Wissensbaumes er gespeichert liegt.

```
# ~ encoding: utf-8 ~

class Part(object):

    def __init__(self, name, path=
        ""):
        self.name = name
        self.path = path
        self.properties = []

    def add_property(self, prop,
        value):
        """
        """
        self.__setattr__(prop,
            value)

    def get_property(self, prop):
        """
        """
        return self.
            __getattribute__(prop)
```

Um den Mikro-/Makro-Kosmos (beziehungsweise die Teil-/Ganzes-Hierarchie) abbilden zu können, müssen auch die Parts, welche in dem aktuellen Part existieren, gespeichert werden. Hierzu bietet sich ein „Dictionary“, also eine Liste von Schlüssel-Wert-Paaren an. Außerdem sollen die Daten persistent in der objektorientieren Datenbank „ZODB“ gespeichert werden. Gegenwärtig erbt die Klasse `Part` von der allgemeinen Python-Superklasse `object`. Um die Persistenz zu gewährleisten, sowie um die Klasse wie ein „Dictionary“ verwenden zu können, wird statt von `object` von den Klassen `Persistent` und `UserDict` geerbt. Um auf die Parts speichern zu können, muss das Klassen-Attribut `data` als Dictionary vorliegen. Sind all diese Anforderungen implementiert, sieht die Klasse wie folgt aus:

```
1  # ~ encoding: utf-8 ~
2  import persistent
3  from UserDict import UserDict
4
5
6  class Part(persistent.Persistent,
       UserDict):
7
8      def __init__(self, name, path=
           ""):
9          self.name = name
10         self.path = path
11         self.properties = []
12         self.data = {}
13
14     def add_property(self, prop,
           value):
15         """
16         """
17         self.__setattr__(prop,
               value)
18         self.properties.append(prop
               )
19
20     def get_property(self, prop):
21         """
22         """
23         return self.
               __getattribute__(prop)
```

Diese 23 Zeilen reichen aus um den doppelthierarchischen Ansatz in Python zu nutzen. Durch das Erben von `Persistent` ist es auch möglich, sie in der „ZODB“ abzuspeichern. Die Dateien „part.py“ und „start.py“ sind im Anhang zu finden. Bei der „start.py“ handelt es sich um einen Webservice, welcher es erlaubt `Part`-Objekte dynamisch anzulegen und abzurufen.

3.2 Weitergehende Meta-Programmierung in Python

Mit der zuvor vorgestellten Part-Klasse, sowie weiteren, in diesem Kapitel erläuterten Möglichkeiten der Meta-Programmierung ist es möglich, eine äußerst dynamische Anwendung zu erstellen. Python bietet neben dem dynamischen Hinzufügen von Attributen noch die Möglichkeit zur Laufzeit Funktionen, Klassen und Module zu implementieren. Bei solchen Objekten spricht man anschließend von „synthetischen“ Funktionen/ Klassen/
Modulen. Diese drei Möglichkeiten werden im Folgenden kurz vorgestellt und sind allesamt (Wissen sowie abgeänderte Beispiele) den Vortrag „Python Metaprogramming for Mad Scientists and Evil Geniuses“ von Walker Hale entnommen[7].

Synthetische Funktionen

In vielen Scriptsprachen beziehungsweise dynamischen Programmiersprachen ist es möglich, Programmcode aus Strings direkt einzulesen. In JavaScript ist dies zum Beispiel über die Funktion `eval` möglich. In Python wird dafür `exec` verwendet. Im Folgenden ein Beispiel:

```
1  >>> d = {}
2  >>> exec """\
3  ... def say_hello_to(name):
4  ...     print("Hello %s" % (name))
5  ... """ in d
6  >>> d.keys()
7  ['__builtins__', 'say_hello_to']
```

```
>>> say_hello_to = d['say_hello_to']
>>> say_hello_to("Bob")
Hello Bob
```

Hierbei wurde die Funktion `say_hello_to` erstellt und anschließend demonstriert. Allerdings sollte man hierbei aufpassen: Da Funktionen aus einem String erzeugt werden, können sie ein enormes Sicherheitsrisiko darstellen!

Synthetische Klassen

In dem Vortrag von Walker Hale wird klar, dass eine Klasse in Python nichts anderes ist als ein „Dictionary“, welches auf Funktionen zeigt. In Python gibt es die Funktion `type()`, welche den Typen eines Objektes zurück gibt. Diese kann allerdings auch dazu genutzt werden, generisch neue Klassen zu erstellen:

```
>>> def __init__(self, name):
...     self.name = name
...
>>> def say_hello_to(self, name):
...     print("%s says hello to %s" % (self.name, name))
...
>>> d = dict(__init__=__init__, say_hello_to=say_hello_to)
>>> SynteticClass = type('SynteticClass', (), d)
>>> obj = SynteticClass('Tim')
>>> obj.say_hello_to('Bob')
Tim says hello to Bob
```

Im eben gezeigten Beispiel wurden zwei Funktionen erzeugt und in einem „Dictionary“ gespeichert. Anschließend wurde daraus eine Klasse erstellt, instanziiert und die Funktion `say_hello_to` ausgeführt.

Auf diese Weise ist es möglich, die Part Klasse dynamisch zu erweitern.

Synthetische Module

Mit Hilfe des `new` Modules ist es möglich, zur Laufzeit neue Python-Module zu erzeugen. Anschließend kann man sie mit Hilfe des `sys` Modules registrieren und anschließend normal importieren. Dies geht wie folgt von statten:

```
>>> import new, sys
>>> my_module = new.module('say_hello_module', 'A fake module.')
>>> def say_hello_to(name):
...     print("Hello %s" % (name))
...
>>> my_module.say_hello_to = say_hello_to
>>> sys.modules['say_hello_module'] = my_module
>>> del new, sys, say_hello_to, my_module
>>> my_module.say_hello_to("Bob")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'my_module' is not defined
>>> import say_hello_module
>>> say_hello_module.say_hello_to("Bob")
Hello Bob
```

Auch hier sollte man aufpassen: auf diese Weise ist es möglich, bestehende Module zu überschreiben. Bei sogenannten Monky-Patches (Bezeichnung für das Patchen von Code zur Laufzeit) ist dieser Effekt sogar erwünscht: Man bekommt die Möglichkeit bestehende Module zu bearbeiten, ohne ihren Code ändern zu müssen.

3.3 Warum Metaprogrammierung?

Nachdem nun ausführlich die Möglichkeiten der Metaprogrammierung in Python dargelegt wurden, stellt sich die Frage: Warum oder wofür Metaprogrammierung? Wie bereits zuvor genannt worden ist, stellt

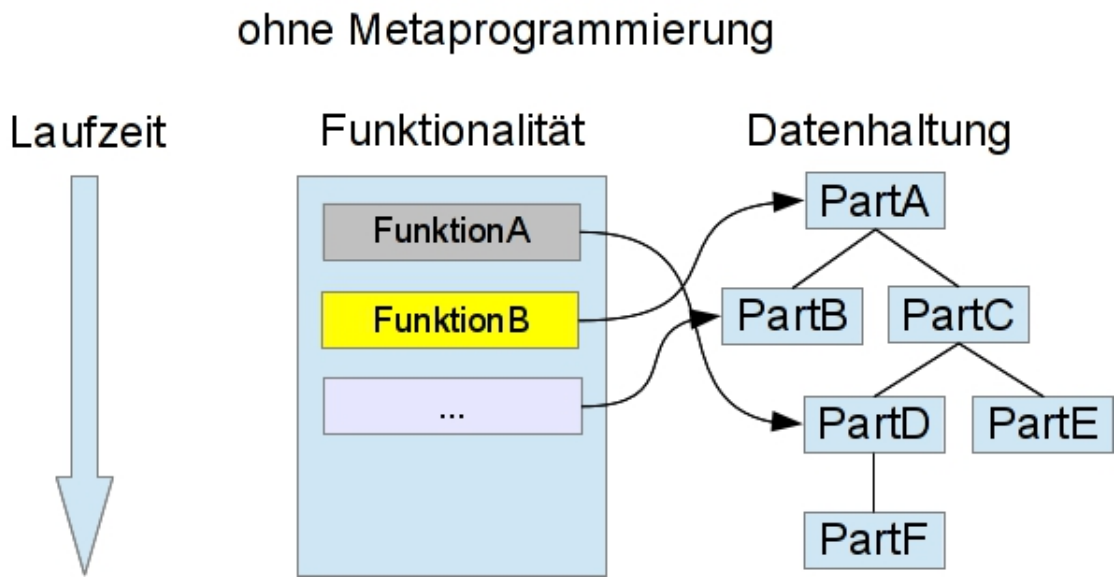

Abbildung 2: Schematischer Programmablauf ohne Metaprogrammierung

die hierarchische Struktur der Datenhaltung das besondere an CYBOP dar.

Wenn nun Programmcode auf die Datenstruktur zugreift, geschiet dies weitestgehend losgelößt von der hierarchischen Struktur. Funktionen aggieren in diesem Fall nur als Wrapper. Also als Programmcode, welcher die Instanzen der `Part` Klasse umschließt und so Funktionalität auf ihr ausführt. Die Funktionalität liegt allerdings nicht selbst bei den Instanzen. Abbildung 2 zeigt dies schematisch. Mit Hilfe der Metaprogrammierung ist es unter anderem möglich, dynamisch Funktionen an Instanzen von Objekten zu fügen. Diese Funktionen stehen anschließend nicht der gesammten Klasse sondern lediglich einzelnen Instanzen zur Verfügung. Durch Metaprogrammierung kann das Ziel erreicht werden, nicht nur die Datenhaltung, sondern auch den Programmablauf hierarchisch darzustellen. Dies wird durch Abbildung 3 dargestellt.

4 Zusammenfassung

Diese Arbeit soll eine Handreichung zum besseren Verständnis des cybernetischorientierten Ansatzes nach Christian Heller darestellen. Dazu beschäftigte Dokument im Kapitel 2 mit den theoretischen Grundlagen dieses Programmieransatzes. Dies geschiet, indem es zu erst einen Blick auf die Herleitung des „hierachischen Modelles“ verschaft. Im zweiten Schritt erläutert es das verwendete „Schema“ genauer und fügt beides abschließend als „Doppelhierarchie“ zusammen.

Das Kapitel 3 beschäftigt sich mit einer möglichen Umsetzung des doppelthierachischen Ansatzes in der Programmiersprache Python. Dazu wird im Kapitel 3.1 kurz das dynamische Hinzufügen von Attributen erklärt und wie daraus eine generische Klasse entsteht. Anschließend wird in Kapitel 3.2 weitergehende Möglichkeiten der Metaprogrammierung in Python erklärt um schlussendlich in Kapitel3.3 kurz zu erläutern, das dank der Metaprogrammierung die Funktionalitäten direkt in den Wissensbaum hinzugefügt werden können, statt sie nebenher zu verwenden. Dies ermöglicht einen weitaus dynamischeren Ansatz, welcher näher an der Programmiersprache CYBOL ist.

Dieses Dokument hilft nicht nur Entwicklern, welche sich neu mit dem Programmieransatz CYBOP auseinander setzen dessen theoretischen Ansatz zu verstehen, sondern

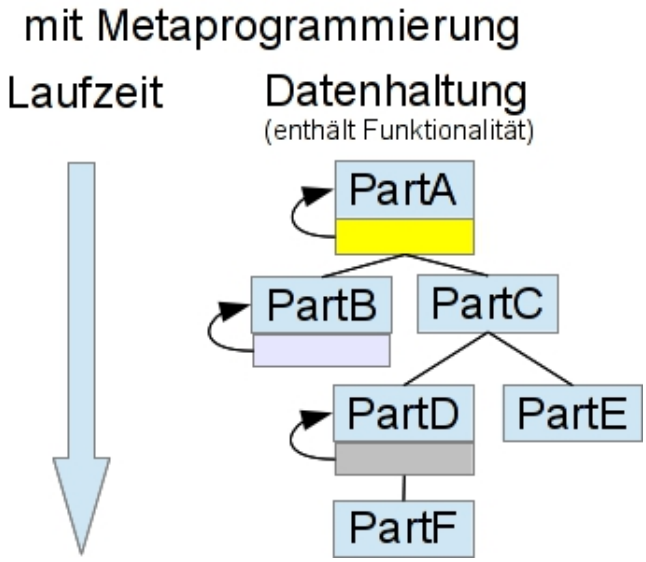

Abbildung 3: Schematischer Programmablauf mit Metaprogrammierung

zeigt auch, das dieser nicht nur an die Sprache CYBOL gebunden ist.

Literatur

[1] Dr. Christian Heller. Svn Repository des CYBOP Projektes, `http://savannah.nongnu.org/svn/?group=cybop`.

[2] Dr. Christian Heller. *Cybernetics Oriented Programming*. Tux Tax, 2006.

[3] Wikipedia Foundation Inc. Wikipedia - Lotuseffekt, `http://de.wikipedia.org/wiki/Lotuseffekt`.

[4] Zope Foundation. Zodb - Zope Object Data Base, `http://www.zodb.org/`.

[5] VERSANT CORP. db40, `http://www.db4o.com/`.

[6] Zope Foundation. Zope, `http://www.zope.org/`.

[7] Walker Hale. Python Metaprogramming for Mad Scientists and Evil Geniuses, `http://www.youtube.com/watch?v=Adr_QuDZxuM`.

Anhang

part.py

```
# ~ encoding: utf-8 ~
import persistent
from UserDict import UserDict


class Part(persistent.Persistent, UserDict):

    def __init__(self, name, path=""):
        self.name = name
        self.path = path
        self.properties = []
        self.data = {}

    def add_property(self, prop, value):
        """
        """
        self.__setattr__(prop, value)
        self.properties.append(prop)

    def get_property(self, prop):
        """
        """
        return self.__getattribute__(prop)
```

start.py

```
1
2  # ~ encoding: utf-8 ~
3
4  from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
5  import cgi
6
7  from ZODB.FileStorage import FileStorage
8  from ZODB.DB import DB
9  import transaction
10
11 from part import Part
12
13 server = None
14
15
16 class CybopHTTPServer(HTTPServer):
17
18     def __init__(self, (HOST, PORT), Handler):
19         self.storage = FileStorage('Data.fs')
20         self.db = DB(self.storage)
21         self.connection = self.db.open()
22         self.root = self.connection.root()
23         HTTPServer.__init__(self, (HOST, PORT), Handler)
24
25
26 class CybopHTTPHandler(BaseHTTPRequestHandler):
27
28     def do_GET(self):
29         """
30         """
31         path_stack, params = self._get_path_stack()
32         if len(params) == 1 and 'name' in params:
33             self.send_response(301)
34             redirect_path = '/'.join(path_stack) + '/' + params['name'].
                   lower()
35             self.send_header("Location", redirect_path)
36             self.end_headers()
37             pass
38         elif path_stack[0] == '':
39             self.return_root()
40         elif path_stack[0] == 'favicon.ico':
41             return
42         else:
43             context, remaining_stack = self._travers_object(path_stack)
44             if not remaining_stack:
45                 self.return_information_view(context)
46             elif len(remaining_stack) == 1:
47                 self.return_add_view(context)
48             else:
49                 self.redirect_method(context)
50
51     def do_POST(self):
```

```
        """
        """
        path_stack, params = self._get_path_stack()
        context, remaining_stack = self._travers_object(path_stack)
        params = cgi.FieldStorage(
            fp=self.rfile,
            headers=self.headers,
            environ={'REQUEST_METHOD': 'POST',
                     'CONTENT_TYPE': self.headers['Content-Type'],
                     }
        )
        new_part = Part(remaining_stack[0].title(), path=self.path)
        for param in params:
            if param[:5] == 'value':
                pass
            else:
                number = param[-1:]
                if 'value' + number in params:
                    new_part.add_property(params[param].value, params['value'
                        + number].value)
        context[remaining_stack[0]] = new_part
        self.send_response(201)
        self.send_header("Location", '/')
        self.end_headers()
        transaction.commit()

    def return_information_view(self, context):
        """
        """
        heading = '<h1>%s</h1>\n<h2>%s</h2>' % (context.name, context.path)
        child_links = ["<li><a href='%s'>%s</a></li>" % (
                context[key].path,
                context[key].name
            ) for key in context.keys()]
        body_text = ["<b>%s:</b><p>%s</p>" % (prop, context.get_property(
            prop)) for prop in context.properties]
        message = """%s\n
        %s\n
        <h3>All properties:</h3>
        %s\n

        <h3>All Subpages:</h3>\n
        <ul>\n%s</ul>
        """ % (heading, self._add_new_form(), '\n'.join(body_text), '\n'.
            join(child_links))
        self.send_response(200)
        self.end_headers()
        self.wfile.write(message)
        return

    def return_add_view(self, context):
        """
```

```
        """
        self.send_response(200)
        self.end_headers()
        heading = "<h1>Add a new Part at the path %s</h1>" % (self.path)
        add_form = """
        <p>
          You can now add your information to the Part.
        </p>
        <form method="POST">
          <table>
          <tr>
            <td><b>attribute name<b></td>
            <td><b>attribute value</b></td>
          </tr>
          <tr>
            <td>
              <input name="attribute1" type="text" size="30"/>
            </td>
            <td>
              <textarea name="value1" type="textarea" cols="30" rows="4"></
                  textarea>
            </td>
          </tr>
          <tr>
            <td>
              <input name="attribute2" type="text" size="30" />
            </td>
            <td>
              <textarea name="value2" type="textarea" cols="30" rows="4"></
                  textarea>
            </td>
          </tr>
          <tr>
            <td>
              <input name="attribute3" type="text" size="30" />
            </td>
            <td>
              <textarea name="value3" type="textarea" cols="30" rows="4"></
                  textarea>
            </td>
          </tr>
          <tr>
            <td>
              <input name="attribute4" type="text" size="30" />
            </td>
            <td>
              <textarea name="value4" type="textarea" cols="30" rows="4"></
                  textarea>
            </td>
          </tr>
          <tr>
            <td>
```

```
            <input name="attribute5" type="text" size="30" />
          </td>
          <td>
            <textarea name="value5" type="textarea" cols="30" rows="4"></
                textarea>
          </td>
        </tr>
        </table>
        <input type="submit" value=" Add ">
      </form>
      """
      self.wfile.write("%s\n%s" % (heading, add_form))
      return

  def redirect_method(self, context):
      """
      """
      self.send_response(301)
      if context == self.server.root:
          self.send_header("Location", '/')
      else:
          self.send_header("Location", context.path)
      self.end_headers()
      pass

  def _travers_object(self, path_stack, active_object=None):
      active_element = None
      if not active_object:
          active_object = self.server.root
      while path_stack != []:
          active_element = path_stack.pop(0)
          if active_element in active_object:
              active_object = active_object[active_element]
              active_element = None
          else:
              break
      if active_element:
          remaining_stack = [active_element] + path_stack
      else:
          remaining_stack = path_stack
      return active_object, remaining_stack

  def return_root(self):
      heading = '<h1>Welcome to the Cybernetic Oriented Webservice</h1>'
      child_links = ["<li><a href='%s'>%s</a></li>" % (
             self.server.root[key].path,
             self.server.root[key].name
          ) for key in self.server.root.keys()]
      message = "%s\n%s<h3>All Subpages:</h3>\n<ul>\n%s</ul>" % (heading,
          self._add_new_form(), '\n'.join(child_links))
      self.send_response(200)
      self.end_headers()
```

```
199         self.wfile.write(message)
200         return
201 
202     def _get_path_stack(self):
203         path_stack = self.path.split('/')
204         path_stack.pop(0)
205         if path_stack == [''] or '?' not in path_stack[-1]:
206             return path_stack, {}
207         else:
208             params = {}
209             stack_rest, param_string = path_stack.pop(-1).split('?')
210             if stack_rest != ['']:
211                 path_stack.append(stack_rest)
212             param_list = param_string.split(',')
213             for param_pair in param_list:
214                 attribute, argument = param_pair.split('=')
215                 params[attribute] = argument
216             return path_stack, params
217 
218     def _add_new_form(self):
219         return """<form>
220         Add new Subpage with name: <input name="name" type="text" size="30"
              /> <input type="submit" value=" Ok ">
221         </form>
222         """
223 
224 if __name__ == '__main__':
225     try:
226         HOST, PORT = "localhost", 9999
227         server = CybopHTTPServer((HOST, PORT), CybopHTTPHandler)
228         print("\n*************************************************************\
              n" + \
229             "***                                                   ***\n" +\
230             "***  This Server handels WebRequests and works with a ***\n" +\
231             "***  Cybernetics Oriented Database Model in the   ***\n" +\
232             "***  Backend.                                         ***\n" +\
233             "***                                                   ***\n" +\
234             "***  Starting TCPServer on %s:%s          ***\n" % (HOST, PORT)
                  + \
235             "***  To shut down, press <CTRL-C>                     ***\n" +\
236             "***                                                   ***\n" +\
237             "*************************************************************\n"
238         )
239         server.serve_forever()
240     except KeyboardInterrupt:
241         pass
242     finally:
243         server.connection.close()
244         server.db.pack()
245         server.db.close()
246         server.storage.close()
```