Constructing Complex Things without Getting Confused - Programming Techniques and Reduction of Cognitive Load

Michael Weigend, *michael.weigend@uni-muenster.de*
Institut für Didaktik der Mathematik und der Informatik, University of Münster, Germany

Abstract

A central issue in practical informatics is coping with complexity. This paper discusses three programming techniques, which are strongly connected to the idea of structuring complex problems and which can be used to reduce working memory load: smart naming, overloading operators and implementing transparency. These techniques can be integrated in constructive classroom projects. The goal is not just to practice specific programming skills but to develop computational thinking (especially the ability to structure) which is useful in many areas.

According to Baddeley's theory of working memory, the part of the cognitive system managing information processing during problem solving has a very limited storage capacity. Each part of a computer program can be seen as an external aid used to reduce cognitive load. Programmers use the program text they have already written, to step on in the development of software. People also use formal program text to document algorithmic ideas and explain them to other persons.

There are several methods to increase the readability of program texts. Since working memory can only store textual information, which can be articulated in two seconds (Baddeley 1998), short names for objects are essential. Polymorphism in object-oriented programming can be used to adopt a familiar concept of activity to a new domain. For example, since childhood we have developed a rather abstract concept of addition, which we can use to create all kinds of algorithms. By overloading the plus operator (+) it is possible to extend the concept of addition to new areas. Adopting existing concepts makes it unnecessary to rehearse and internalize new concepts. Paradoxically we use object oriented programming for overloading operators in order to avoid object oriented *thinking* on a higher level of problem solving.

A third and rather technical method of reducing cognitive load is making necessary but complex activities completely transparent. For example in Python you can define so called *properties* in order to hide the implementation of safe access to object attributes.

All examples in this paper are based upon the programming language Python, which might be called Logo-like for several reasons, mainly because it is easy to learn (low threshold) and supports developing cognitive skills by active programming.

Keywords

programming; naming; polymorphism; Python; working memory

Introduction

Students learn programming at school not (necessarily) because they want to or are supposed to become professional software developers. Many pedagogues have the feeling that programming leads to fundamental competences that are useful in many areas of the lives of future members of knowledge societies and that everybody should learn (e.g. Guzdial, Wing, Schwill).

One of these computational competencies is structuring a complex problem into smaller coherent parts. It is a facet of “computational thinking” (Wing) and a “master idea” of informatics (Schwill). According to Seymour Papert, constructing things is an opportunity to discover “powerful ideas” (Papert, 2000). Just telling such ideas is not enough. The power of powerful ideas has to be *experienced* within a context of activity to be truly understood. Papert uses the example of “probability” (one might also say nondeterminism) to illustrate this (Papert 2000). Imagine a student developing a light-seeking Lego robot. A Logo program controlling its movement compares the outputs of two light sensors. When the light source is more to the left, the vehicle turns to the left, and otherwise to the right. A problem occurs, when there are flat obstacles on the ground, which do not block the line of sight to the light source. The vehicle might stop with turning wheels and never reach the light. Nondeterminism helps to solve this problem. The control program could be extended by a command like this:

```
With probability p turnleft x units
```

Basically this means, that sometimes surprisingly the vehicle does something strange: It just goes to the left. In case it is blocked by an obstacle in that very moment, this arbitrary movement helps to get free again and finally find the way to the light. In case it is not blocked this movement leads to a more erratic path, but (if p and x are small enough) it does not hinder the vehicle from going to the light. In Papert’s words the idea of “probability” is *powerful in its use*, because students can experience that it is useful to solve real problems. Additionally this idea is *powerful in its connections*, which means that can be used in several different domains. For example “probability” is essential to explain evolution. And finally it is *powerful in its roots*, since the comprehension of this idea is based upon intuitive knowledge, kids already know, making them feeling powerful.

In this contribution I am going to discuss ideas connected to a central aspect of problem solving, namely the reduction of working memory load by applying certain techniques like naming, polymorphism and transparency.

Working memory and computer programming

We need structuring because the capacity of our working memory is very limited. During problem solving our mind processes chunks of information very quickly and partly in parallel. In moments of “hard thinking” we focus attention to just a few things. The part of the cognitive system managing this, is called working memory (Baddeley 1998, 2003, Dehn 2008). According to Baddeley’s model, working memory consists of several subcomponents for storing verbal and visual information: the Phonological Loop and the Visuospatial Sketchpad. The Phonological Loop is analogue to an audiotape with very limited length. It is able to store verbal content (like variable names and function names), which can be articulated by the subject in at most two seconds. That implies for example that word length affects memory span. You can handle more variable names when they are short. The Visuospatial Sketchpad is supposed to have similar limitations but these have been less well assessed yet.

One of the consequences of the small storage capacity of working memory is that we have to develop big knowledge structures “piece by piece” using external aides like pictures, diagrams and text documents. The process of computer programming can be considered as building up a

system of external aides for further development. Basically you start with a few lines of code. When you are sure they work correctly, you go on and add more lines of code using the existing text as external aid.

Thus, readability of the text is essential for development progress and not just for maintaining existing programs. The quality of the language style (including class structure and naming) is checked all the time during development and usually needs improvement from time to time. When you get stuck it might be caused by an inefficient external aid for your working memory. This point is stressed especially in Extreme Programming (Beck 1999). Refactoring the whole program in order improve its technical quality (including readability) is a feature of this methodology. While efficiency aspects can be handled in an isolated "tuning phase" at the end of a project, readability of already existing code is crucial *in each phase* during the development. A bad structure or missing naming conventions is a burden which might slow down the development speed or – especially in case of novice programmers – even block the process totally.

Smart naming

"Code is read much more often than it is written" (Guido van Rossum)

Style guides give much advice how to name objects. 30% of the Python style guide is about naming conventions (van Rossum 2009). This includes rules like "use capitalized identifiers as class names and non-capitalized identifiers for attributes, methods and class instances" or "a name should express the role or meaning of the named object within the algorithmic context".

Some rules like the first example represent some traditions within the programmers' community, but nothing more. Still, these conventions increase readability. The second example goes deeper. A meaningful name like *count* or *sum* represents a chunk of algorithmic information and thus helps to understand a program. When you read the word sum you immediately think of addition and the concept that the sum of some numbers is stored in this variable. It is of advantage that you already know the concept of a sum. There exist some very short algorithms, which are still difficult to understand, because the identifiers do not represent well known concepts.

Explicit naming implies structuring. You do not use an existing name and refer to an object (one step) but you introduce a name first and use the new name then (two steps). This might lead to an immense reduction of cognitive load. Example: The following lines of Python code calculate the size of a window, which is part of a house.

One step (no naming):

```
size = (house.floors[floor_nr].windows[window_nr].height *
        house.floors[floor_nr].windows[window_nr].width)
```

Three steps (introducing names):

```
window = house.floors[floor_nr].windows[window_nr]
height = window.height
width = window.width
size = width * height
```

By introducing the names `window`, `width` and `height`, the reader is supported to divide the process of verifying the logical correctness in three independent parts. Thus, introducing names is a kind of semantic preprocessing. The program text is written taking into account that the working memory capacity of a future reader is limited. Moreover, the *programmer* can use short names like *height* and *width* instead of complex references to develop an algorithm. During the problem solving process it is now possible to keep *additional* relevant chunks of information in

working memory and process them in order to find a solution. Structuring by naming is also common in everyday language: "I have got a Renault Megane Scenic, built in the year 2001. For *this car* I need a new headlight and a screen wiper blade." "Tim is my brother's youngest daughter's dog. Yesterday I saw *Tim* alone on the road to the forest."

High school students learn how to use names for modeling and problem solving in science: "Let c be the concentration of hydrochloric acid in a specimen…" These examples suggest that smart naming is not just a technical skill of specialists but part of general competences like problem solving and communication.

In summer 2009 I conducted a study on the question to what extent students without former computer science education are capable to apply naming techniques, when they have to write algorithms (Weigend 2010). One type of workshop was about referring to objects within a complex three-dimensional structure. In phase 1 the students had to follow given instructions identifying ten broken parts of a fictive power plant on Mars. These instructions contained different types of naming and referring, for example

(1) The pyramid in the right corner of the platform is called *corner pyramid*. Write number 1 on the *corner pyramid*.

(6) When you move from the *corner pyramid* along the edge of the platform to the left, you reach a cube. Write 6 on this cube.

The name corner pyramid (introduced in instruction 1) was later used in instruction 6 making a path-like reference simpler.

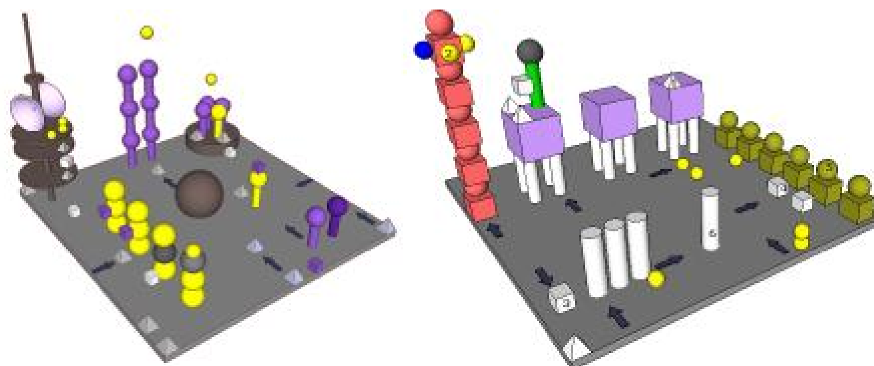

Figure 1. 3D-structures used in "Mission to Mars 3" (Weigend 2010)

Most of the 49 students (grade 9, average age 15.5 years) had no problems to solve this task. In average 95% of the instructions were interpreted correctly. In phase 2 the students had to write similar instructions by themselves. They got a picture showing a different structure, representing a fictive factory on Mars. Six parts were labeled with numbers. The task was to write references to them. Later, the students gave their instructions to classmates who had to identify the parts on an unlabelled picture of the factory. So we had a social situation in which it was important to create readable algorithmic text. The question was: Which of the naming and referring techniques from phase 1 did the students adopt in phase 2?

Only 23 out of 49 students used some kind of naming concept for individual entities. And not more than just three of them explicitly introduced a name (like *corner pyramid*), which they used in other instructions later. Avoiding explicit naming could also be observed in other workshops, where participants had to read and write algorithms for drawing two-dimensional ground plots (Weigend 2010). This suggests that smart naming – which I consider being a facet of

computational thinking – is not just learned en route during normal socialization but needs to be taught explicitly.

What can teachers do to encourage smart naming in a programming project? Since naming is essential in every program, the content or topic of a programming task does not matter. Methodology is more important. Extreme Programming (Beck 1999) seems to be an appropriate approach for classroom projects, since it provokes communication and quick development and leaves space for experimenting and learning (Weigend 2005). Let me just mention four features: (1) Pair programming. Two developers share one computer and work together. Thus, they talk about the evolving program text all the time. Names are not arbitrarily chosen but discussed. (2) Development in short iterations. Students select "stories" (short descriptions of functionality) and implement them. At the end of an iteration (which might take just two lessons) there is a runnable program which can be presented and discussed. (3) Refactoring. From time to time the whole program is restructured. This includes changing names, in order to get a better language quality. In contrast to traditional software engineering this is not seen as an annoying disruption which should be avoided, but as a necessity. (4) Collective ownership of code. Rephrasing program text is made easy because each member of the team is allowed to change each line of code. The team (consisting of several pairs) shares the responsibility for the whole project.

Overloading operators – using familiar concepts in new domains

Structured computer programming implies dividing a complex problem into smaller parts by defining functions and/or classes. According to working memory theory the parts are worthless for further program development unless the developer is really familiar with them. A function must be an intuitive coherent piece of knowledge (chunk). If it is not, the developer has to rehearse its usage until she or he has completely internalized its effect. Otherwise the developer is in a split attention situation (Ayres & Sweller 2005). She or he has to look up the function definition (which is written at a different place within the program) and memorize it explicitly while trying to formulate an appropriate function call.

Andrea diSessa states that there are relatively few abstract intuitive concepts people use for problem solving within unfamiliar domains (diSessa 2001). He calls them phenomenological primitives (p-prims). He observed that even well trained scientists use intuitive concepts (like resistance or friction), when they encounter a problem which is new to them. Adopting an abstract concept to a specific situation seems to be easier than developing a completely new concept. In the next section I am going to discuss the concept of addition as an example.

The concept of addition

People use the concept of adding since kindergarden. In mathematics, adding is an arithmetical operation on numbers. In primary school adding natural numbers is often introduced adopting the metaphor "arithmetic is collecting things" (Lakoff & Nunez 1997). A number is represented by a collection of things of one kind, e.g. beads. The operation 2 + 3 can be visualized like this: put two beads on the table, add another three beads and then count the resulting number of beads on the table. In this domain subtraction means taking away some beads from the table. Another metaphor common in elementary math teaching is "arithmetic is moving along a line". Visualizing the set of real numbers by a horizontal line, the mathematic term 2 + 3 -1 can be represented by this sequence: Move to the right for two steps. Move to the right for 3 steps. Move to the left for one step.

The activities I have just described can be seen as metaphors for an arithmetic operation. But vice versa the arithmetic operation addition can also serve as an abstract model of real life activities like moving, collecting or concatenating.

In informatics this is called polymorphism or – more specific – overloading of operators. Joseph Bergin states that “polymorphism and not the class concept is the big idea of object-oriented programming” (Becker et al. 2001, p. 410). The funny thing is that polymorphism allows the programmer *not* to think object oriented. You have a rather abstract operation in your mind – in the world of OOP one might say a type of message – and use it for algorithm development. The operation exists *independent from objects*. The details of the execution are of secondary importance, they are implemented within the class definition and you need not to think about them while developing an algorithm. Instances of different classes may react in a slightly different way to the same message, but the meaning of the message is – on an abstract level – the same.

Overloading operators in Python

In Python operators and built-in functions can be overloaded by defining “magical” methods with certain names starting and ending with two underscores. The following Python statement defines a simple class modeling length (example taken from Weigend 2010a).

```
class Length(object):
    meter = {'mm': 0.001, 'cm':0.01, 'm':1, 'km':1000,
                'in':0.0254, 'ft':0.3048,
                'yd':0.9143, 'mil':1609}                 #1

    def __init__(self, value, unit):
        self.value = float(value)
        self.unit = unit

    def getMeter(self):
        return self.value * self.meter[self.unit]

    def __add__(self, other):                            #2
        s = self.getMeter() + other.getMeter()
        return Length(s/self.meter[self.unit], self.unit)
    ...

    def __repr__(self):
        return str(self.value)+' ' + self.unit           #3
```

An instance represents a length or spatial distance through a float number and a unit like *cm* or *ft*. The class attribute meter (#1) is a dictionary mapping units to the number of meters they represent. For example 1 millimeter is equal to 0.001 meters. Overloading the plus-operator is implemented by defining method `__add__()`. Each time the Python interpreter evaluates an expression like a + b, it sends a message like `a.__add__(b)` to object `a`. In the same way the developer can define further operations, including subtraction, multiplication, division or Boolean functions like <, >, == and so on. The method `__repr__()` returns a printable representation of the object.

Like Logo, Python can be used in an interactive mode. After executing this script the class can be used and tested. Behind the prompt `>>>` you type a single statement. It is executed immediately after you have hit the ENTER-key. Any output is displayed in the subsequent line. This is an example dialog:

```
>>> foot = Length (1, "ft")
>>> earth_diameter = Length(12713.507, "km")
>>> earth_diameter + foot
12713.5073048 km
```

```
>>> Length(2, "cm") + Length(2, "in")
7.08 cm
```

The concept of adding in the abstract meaning of "putting together" is also used in the context of blending substances. We *add* sugar to lemon juice to make it taste sweeter and we use the plus-operator in chemical reaction equations. Instances of the following Python class represent potions, consisting of different ingredients. The content of a potion is stored in a dictionary mapping the name of an ingredient (like lemon juice) to the amount to which it is contained (number of grams).

```
class Potion(object):

    def __init__(self, ingredient=None, grams=None):
        if ingredient:  self.content = {ingredient:grams}
        else: self.content = {}                          # empty
dictionary

    def __add__(self, other):
        result = Potion()
        result.content = self.content.copy()
        for i in other.content:
            if i in self.content:
                result.content[i] += other.content[i]     # update
ingredient
            else: result.content[i] = other.content[i]    # new
ingredient
        return result

    def __repr__(self):
        c = self.content
        s = "Substance consisting of \n"
        for i in c:
            s += i + "("+str(c[i])+ " g)\n"
        return s
```

In the following dialogue with the interactive Python shell, I make lemonade in three steps and then display the result. Note that the meaning of the +-operator in this context is quite easy to comprehend without any knowledge about the class definition. Actually, this formal program text could be used to explain another human how to make lemonade.

```
>>> sparkling_water = Potion("water", 998) + Potion("carbon dioxide",
2)
>>> syrup = Potion("lemon juice", 50) + Potion("sugar", 100) +
    Potion("water", 100)
>>> lemonade = sparkling_water + syrup
>>> lemonade
Substance consisting of
water(1098 g)
carbon dioxide(2 g)
lemon juice(50 g)
sugar(100 g)
```

Models of blends are useful in many areas. In Witten, the city where I live, there is a steel company producing high quality steel from scrap. Basically, the electric arc furnace is charged with different kinds of scrap in such a clever way that the resulting blend has the exact chemical

composition of the target product. A computer program calculates the required amounts of various scrap (and pure alloy). Such program should contain a class modeling the chemical composition of scrap charges similar to the class `Potion` above.

Dealing with fuzziness

Intuitive concepts like "addition" are fuzzy. This is an implication of being abstract. Abstraction (Latin abstrahere = to take away or to remove) means that you focus on the important and ignore the unimportant. Abstraction is essential when you process a concept in your working memory while creating an algorithmic idea on a high level. But when it comes to implementation as a runnable computer program you have to explicate the ignored details in further steps of development. In this perspective, overloading an operator means removing the fuzziness of a familiar (intuitive) operation you are applying to a new domain.

Figure 2 shows two screenshots from an application developed with Python, which evaluates a voting by raising voting cards. First the user loads a picture file (in figure 2 you see a simplified voting situation: yellow cards on a gray carpet). Then she or he clicks on a voting card on the picture to tell the program the color of the card. This color is shown on a label at the bottom of the application window. Due to varying illumination the color of the voting cards on the picture is slightly different. In the next step the user has to define a tolerance for the color recognition using a slide. The selected color is split in a darker and a lighter version (see right screen shot). After pressing the button with the raised hand, the program counts the voting cards and displays the result.

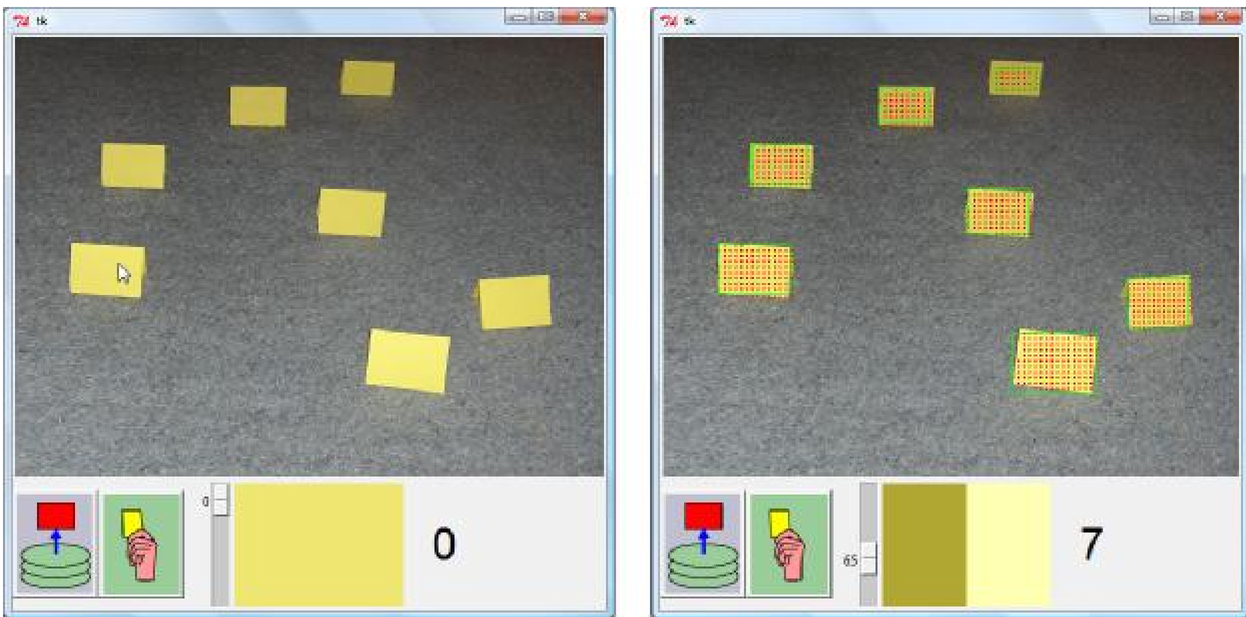

Figure 2. Screenshots from a graphical Python program, which counts voting cards.

The whole Python program consists of 200 lines of code and is too long to be discussed completely here. I focus on a few lines using overloaded operators. The algorithm of counting is mainly implemented in this iteration:

```
c = Cards()                                                 #1
for (x, y) in self.grid:                                    #2
        if self.__get_color(x, y) == self.color:            #3
            self.pic.put("{red red} {red red}", to=(x,y))   #4
            c.extend(Dot(x, y))                             #5
```

#1: This represents a list of voting cards.

#2: The list `self.grid` contains pairs of numbers `(x, y)` indicating pixels of the photo which have a certain distance to each other. (This is to get a better performance. We need not to check every pixel).

#4: Draw a red dot on the picture. So the user can see that the program has found a pixel probably belonging to a voting card.

#5: Update the list of voting cards c. This goes like this: If the pixels is at the border of an already found voting card, this card is extended now including the area of the new pixel. Otherwise a new voting card is created covering just the area of the new pixel.

I did not write a comment to line #3. However, do you understand its meaning? Probably you do. The meaning is: " If the color of current pixel is the color of voting cards, …"

Suppose the voting cards are yellow. In this case we can say: "If the current pixel is yellow, it probably belongs to a voting card, and we have some business to do." This is a comprehensible idea. It is simple. Still, it represents the very core of a rather complex algorithm. The idea is comprehensible because it is a combination of just a few chunks of information, which we can process in our working memory. But the prize for simplicity is fuzziness. For example, what does it mean to claim "the pixel is yellow"? In line #3 two objects representing colors are compared by means of the equal-operator ==. In this context a fuzzy concept of "being equal" is enough to understand the if-statement. We trust that the method implementing the comparison works in an appropriate way. Obviously, this is not the usual arithmetical concept of equality. The program contains a class representing color-objects. And here the equal-operator == is overloaded by defining a method with the name `__eq__`:

```
def __eq__(self, other):
        t = self.tolerance                  # use a shorter name
        return (abs(self.r - other.r) <= t) and \
                (abs(self.g - other.g) <= t) and \
                (abs(self.b - other.b) <= t)
```

This method returns the Boolean value `true`, if and only if the differences of the rgb-values of the compared color-objects are within a tolerance interval.

Transparency and information hiding

In OOP an object is sometimes described as a fortress keeping most of its internal structure hidden from the public. The attributes are private, nobody in the environment is allowed to read or change them by immediate access via assignments like `object.attribute = newValue`. But the environment may use special methods – called getters and setters – to read or change some attributes. The setters may contain safeguards protecting the internal data from invalid changes.

Consider a class Label, which models labels for articles offered in a shop. A Label-object has two attributes: the price of the article and a text with at most 12 characters. These two attributes are declared as private. But for each of them a setter and a getter method is defined, which allow controlled access. Fig. 3 shows on the left hand side the corresponding UML class diagram.

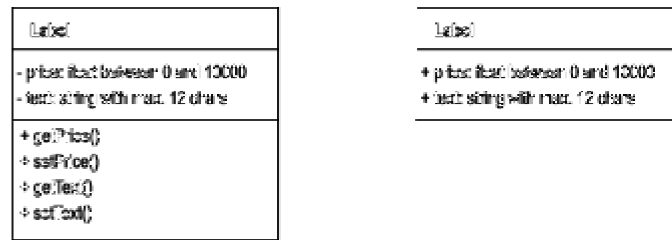

Figure 3. UML class diagrams

The point is that both attributes represent information which is meant for the public and not just for internal calculations. So these attributes are intended to be public. The class diagram with public attributes and without the setters and getters would be much more simple and comprehensible (fig. 3 right diagram). Python offers a mechanism to define attributes, which “look public” from the outside. Still, the access to them is controlled by special private methods. This technique is called properties. The following script illustrates how to use it.

```
class Label:
    def __init__(self):
        self.__price = 0.0                    #1
        self.__text = "no name"

    def getPrice(self):
        return self.__price

    def setPrice (self, x):
         assert 0 <= float(x) < 10000         #2
         self.__price = float(x)              #3

    def getText(self):
        return self.__text

    def setText(self, t):
        try:
            self.__text = str(t)[:12]         #4
        except:
            self.__text = "no name"

    text = property(getText, setText)         #5
    price = property(getPrice, setPrice)
```

The method `getPrize()` returns the present value of the private attribute `__price` (see line #1). Attributes starting with two underscores are private and cannot be accessed directly.

Method `setPrize()` changes the value of the private attribute `__price`. The assert statement (line #2) checks whether or not the argument of the method call is a number between 0 and 10000. If this condition is not true, an exception is raised. That means the program run is aborted and the Python interpreter outputs a message about that. Python does not require typing. Statement #3 contains a casting and makes sure that the private attribute __price gets a floating point number.

In contrast to the first setter, `setText()` does not raise exceptions. It contains a `try` … `except`-statement to prevent abortions caused by inappropriate arguments. Statement #4 guarantees that the attribute text is set to a string with at most 12 characters.

In the last two lines of the script, two so called *properties* are defined. The (public) names `price` and `text` can be used in assignments like regular attribute names. In the following statements I check the effect of the safeguards.

```
>>> t = Label()         # create an instance of the class Label
>>> t.price = 10        # assign the integer object 10
>>> print(t.price)      # the attribute value is a floating number
10.0
>>> t.price = -1        # invalid for Label-objects
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    t.price = -1
  File "…/label.py", line 10, in setPrice
    assert 0 <= float(x) < 10000                                  #2
```

Using properties instead of public `set()` and `get()` methods does not mean overall reduction of complexity but just redistribution of complexity. The access to attributes from the environment of the class becomes simpler. But the class definition becomes more complex. The total complexity cannot be reduced.

The motive for controlling access to attributes is to provide logical safety. A complex system must contain mechanisms that support error detection. For example, some erroneous program component might try to assign a negative price-value to a Label-instance. But this security aspect is completely irrelevant for a programmer, who is developing a routine that just *uses* Label-objects. During the algorithm development it would be an unnecessary burden on working memory. One facet of computational thinking is to be aware of this and to avoid such burdens.

Discovering the poetry of computer programs

Professional computer programmers have to deal with huge class libraries. They use complex development environments with IntelliSense-like functionality, which help to find an appropriate method for a given purpose. One might claim that students should rather learn how to use repositories and integrated development environments (IDEs) like Eclipse, in order to get a more realistic idea of modern programming. But in this case the programming language loses its function as means of expression. A well written program is a document made for humans. Like a diagram or a natural language text, it can be used for creating and communicating intellectual content. For example textbooks on bioinformatics contain programs explaining natural information processing. On the other hand, a complex program, consisting of highly specialized parts, created with the support of semiautomatic code generators is not a well readable document any more. At this point classroom programming – focusing on computational thinking – differs fundamentally from professional software development. Classroom activities at schools and partly even at universities are not a vocational training for future software engineers. In these educational contexts a programming language primarily serves as a means to express ideas rather than as engineering technology to build efficient and secure software.

Let me conclude with two suggestions for teachers, who are organizing constructionist computer science education and intend to create an environment that inspires students to discover the expressive power of programming language constructs.

(1) Encourage students to communicate their algorithmic ideas and programs to other people. The poetry of programming text can only be discovered when students read it and talk about it all the time. A social environment based on the ideas of Extreme Programming might be a good approach. There should be space for experimenting and refactoring.

(2) Create interesting software project ideas based on a model of some aspect of reality (like length, color, blends etc.). Usually such model is a class, of which a rudimentary version might

be prepared by the teacher. The language constructs, which the teacher wants to be learned, are not interesting per se to the students, when they start a project. The genuine motives to construct a product are rooted somewhere in the everyday life of the students. They just want to create something cool. The academic learning – including discovering “big ideas” – takes place en route.

References

Ayres, P. and Sweller, J. (2005) *The Split-Attention Principle in Multimedia Learning*. The Cambridge Handbook of Multimedia Learning, Richard E. Mayer, Ed. Cambridge University Press, pp. 135–146.

Baddeley, A. (1998) *Recent Developments in Working Memory*. Current Opinion in Neurobiology, Vol. 8/2, pp. 234—238.

Baddeley, A. (2003) *Working Memory Looking Back and Looking Forward.* Nature Reviews Neuroscience, Vol. 4, pp. 829—839.

Beck, K. (1999) *Extreme Programming Explained.* Addison Wesley.

Becker, B. W.; Rasala, R.; Bergin, J; Shannon, Ch.; Wallingford, E. (2001) *Polymorphic Panalists*. Proceedings of the 32nd SIGCSE technical symposium on Computer Science Education. Charlotte, pp. 410−411.

Dehn, M. J. (2008) *Working Memory and Academic Learning*. John Wiley & Sons, Hoboken, New Jersey.

Guzdial, M. (2008) Paving the Way for Computational Thinking. Communications of the ACM Vol. 52 No. 8, pp.25—27.

Lakoff, G., & Núnez, R. E. (1997). *The Metaphorical Structure of Mathematics: Sketching Out Cognitive Foundations for a Mind-Based Mathematics.* In L. D. English (Ed.) Mathematical Reasoning. Analogies, Metaphors, and Images, Lawrence Erlbaum Associates, pp. 21–92.

Papert, S. (2000) *What's the big idea? Toward a pedagogy of idea power.* IBM SYSTEMS JOURNAL, Vol. 39, pp. 720−729.

Schwill, A. (1993): *Fundamentale Ideen der* Informatik *[Fundamental ideas of Computer Science].* Zentralblatt der Mathematik 20, pp. 20–31.

diSessa, A. A. (2001) *Changing Minds. Computers, Learning, and Literacy*. MIT Press, Cambridge, Massachusetts.

van Rossum, G. (2009) *Style Guide for Python Code* (PEP 8, version 68852), (last modified 2009), http://www.python.org/dev/peps/pep-0008/

Weigend, M. (2005) *Extreme Programming im Klassenraum [Extreme Programming in the classroom].* INFOS 2005, München, Germany, http://www.fernuni-hagen.de/schulinformatik/xp_weigend.pdf.

Weigend, M. (2010) *Mission to Mars. A Study on Naming and Referring*. In ISSEP 2010 Proceedings, J. Hromkovic, R. Královiè, and J. Vahrenhold (eds), LNCS 5941, Springer, Heidelberg, pp. 185−196.

Weigend, M. (2010a) *Objektorientierte Programmierung mit Python 3. [Object Oriented Programming with Python 3]*. 4th edition. Bonn, mitp.

Wing, J. M. (2006) *Computational Thinking* Communications of the ACM, Vol. 49, No. 3, pp. 33-35.