Making Computer Science Education Relevant

Michael Weigend

Making Computer Science Education Relevant

Michael Weigend

University of Münster,
Institut für Didaktik der Mathematik und der Informatik,
Fliednerstr. 21, 48149 Münster, Germany
michael.weigend@uni-muenster.de

Abstract. In addition to algorithm- or concept-oriented training of problem solving by computer programming, introductory computer science classes may contain programming projects on themes that are relevant for young people. The motivation for theme-driven programmers is not to practice coding but to create a digital artefact related to a domain they are interested in and they want to learn about. Necessary programming concepts are learned on the way ("diving into programming"). This contribution presents examples of theme-driven projects, which are related to text mining and web cam image processing. The development and learning process is supported by metaphorical explanations of programming concepts and algorithmic ideas, experiments with simple programming statements, stories and code fragments.

Keywords: Computer science education, programming, metaphor, text mining, image processing, internet computing, Python.

1 Diving into Programming

Computer science (CS) education at schools is supposed to "introduce the fundamental concepts of computer science" (CSTA, [1]) and foster computational thinking [2], which includes abstraction, modeling, problem solving and creating algorithms using formal language. In contrast to information technology (IT) education, computer science education is not just about using digital tools but about designing software [1]. Programming (the skill of writing a program to a given task) is considered as a new literacy [3] and an important part of general education, since it is creative, constructive and precise [4].

Programming is a problem solving activity and implies a transfer of knowledge to new scenarios. Among other cognitive operations [5], transfer in problem solving requires recognition (of an analogue problem or a well known general pattern), abstraction (finding general structures by focusing the important aspects), mapping (relating familiar concepts to a new scenario), flexibility (in applying a general pattern on a special scenario) and embedment (combining elements to a whole program). Developing programming skills means practicing knowledge transfer by writing programs and solving similar tasks again and again. Consider this programming task:

"Last rainy day. Develop a program for which the input is 365 integers indicating the amount of rain in each day of the year; and the output is the (index of the) last rainy day."[6].

The solution is a special variant of a general pattern, a "max computation", in which all elements of a sequence have to be compared to a given value and a variable eventually must be updated depending on the result of this comparison. Out of 95 Israeli 11-graders, 69 (73%) were able to solve this task without any help after one year of Java programming. [6] assume, that the others (23 %) had difficulties in flexibility and failed to customize a general pattern to the specifics of a new situation. I mention this example just to illustrate that writing a program from scratch without external help is not easy and requires a lot of training and experience. It is a competence that is developed gradually in many exercises. Typical tasks for practicing contain short and precise descriptions of pre- and post-conditions, which make it possible to check the correctness of the solution. For each concept (algorithmic patterns, language constructs) there are many variants of tasks embedded in scenarios from different domains. Diversity is important (to practise transfer-related operations) but the domains can be chosen rather arbitrarily. For practising search algorithms it does not matter, what to search – the last rainy day in a sequence of weather documents or the last phone call from Anna in a collection of telephone call metadata.

Computer science topics listed in curricula represent the teachers' perspective: "To be well-educated citizens in a computing-intensive world and to be prepared for careers in the 21st century, our students must have a clear understanding of the principles and practices of computer science." (CSTA) But these "principles and practices" – as such – are not necessarily interesting for high school students. For example in Germany the requirements for final high school exams include topics like object oriented programming (classes, inheritance, polymorphism, UML) and finite state automata. Probably most 15 or 16 years old students, who have to decide whether or not they take CS classes, do not even understand what these terms mean.

According to the international ROSE study, most young people in Europe and other well developed countries have a positive attitude towards science and technology but they have a problem with school science. "Topics that are close to what is often found in science curricula and textbooks have low scores on the rating of interest" [7]. Science and technology topics are not interesting as such, but they can get fascinating for young people, when they are embedded in a real life context. There are massive differences between genders: Girls like to learn about body and health, boys are interested in violent and spectacular contexts (e.g. chemical explosives). Both genders are especially interested in unusual and mysterious things (most popular topic: The possibility of life outside earth).

The motive for learning programming is not necessarily intrinsic. Someone might be not interested in "the principles and practices of computer science" at all, but gets involved because she or he wants to create something exciting..

Protagonists of constructionism [8, 9] claim that developing digital artefacts is a very intense experience leading to deeper knowledge than just reading text books. Programming is a way to elaborate knowledge. Construct something interesting and learn on the way. This constructionist approach of "theme-driven programming" has some major implications:

- Diving into programming. When the learner starts a project she or he possibly has only little knowledge about programming and must learn a lot in a short time.
- Once-in-your-life-experience instead of repetition. Creating a digital artefact is a rich experience leading to a unique product. Richness implies that many things happen and many circumstances came together to make the project possible: Motivations that were satisfied through the project, an assignment, collaboration with other persons. In contrast to this unique experience, practicing programming implies repetition of similar activities.
- Priority of the artefact. The primary (subjective) goal of the learner is not to practise programming but to create an artefact. Anna has seen something cool and wants to make a similar thing. Opposed to the practicing approach the product has a higher value than the process of implementation.
- Limitation to basic designs. Programs developed in the classroom differ from professional programs. They are implemented as simple as possible.
- Using scaffolds. In contrast to the practicing approach the primary goal of a project is not to gain fluency (it will happen anyway). The project is the reason for going deeper into programming. Exploring new programming techniques requires "just in time" explanations that open the mind.
- Tinkering. Learning by doing requires the possibility to experiment. The learner modifies the code, runs the program and sees the effect.

Some programming languages/environments support "diving into programming". Python has a very "low threshold" which is easy to overcome by beginners. The line

```
print("Hello!")
```

is a valid Python program. In the interactive mode (Python shell) the user can experiment by writing individual statements which are interpreted and executed after having hit the ENTER-key. The result is displayed in the next line:

```
>>> len("Hello!")
6
```

Scratch is a visual programming environment which allows users to "build" scripts by moving block with the mouse on screen (https://scratch.mit.edu/). In this way syntax errors never happen. Children can rather easily create videos, games or animations.

2 How to Support Diving into Programming

A challenge for teachers and text book authors is to create program examples that are relevant (attractive) and easy to implement. A "dive-into" structure for text book units and classroom activities is this:

1. Present a relevant context. The context is an informatics-related theme or field that students consider to be interesting and important. The social aspects of technology are pointed out; its impact on everyday life and the environment are made

aware. Since the interests of young people are diverse, the context should inspire to a variety of concrete projects. For

2. Explain relevant programming concepts and visualize those using metaphors. According to Lakoff [10] metaphors can serve as vehicles for comprehending new concepts. A structural metaphor is a mapping from one domain of knowledge (source) to another domain (target). A variable (target) is a container for data (source). Calling a function (target) is to delegate a job to a specialist (source). Metaphors help understanding new concepts from a target domain, if the source domain is familiar. Another facet of intuitive models is simplicity. Good metaphors represent intuitive models, Gestalt-like mental concepts, which people are very confident about. People use them when they try to understand, develop or explain programs. Programmers may use different metaphors for the same programming concept. For example, a function can be visualized by the metaphor of a factory, which takes data as input, processes them und outputs new data. A different metaphor is a tool changing the properties of an object, which keeps its identity during the process.

3. Give examples for individual statements (not context-related) for hands-on experimenting and elaborating. Novices need to experiment with new commands or functions. Often, just reading the language reference is not enough if you want to be really confident about the meaning.

4. Give a very simple prototype project ("starter project") that can be copied and tested. This can be the starting point for the development of an extended, more sophisticated program. A starter project is supposed to inspire students to do their own project in this field. Scratch users find starter projects for several topics on the Scratch website, a platform where Scratch users can publish their projects. Scratch cultivates "remixing", that is copying, changing and extending projects. For each project the remixes (successors) and preceding projects are documented. In this way ideas are reused but not stolen, since each contributor is mentioned in the history of a piece of software. A problem of remixing is that "blind copying" does not help understanding. Someone might take a program, change a small part and make it look different still not understanding the other parts.

A starter project can initiate a development process in the style of agile programming (Extreme Programming [11]). Students start with a very short program that implements a basic story. They test it and debug it until it works and until it is fully understood. This is the first iteration. Then they add a few lines of code to implement the next story. They develop the project in a couple of very quick iterations and learn on the way step by step. In that way – ideally – both programming competence and the program (the digital artefact) grow in parallel.

It is essential to step on not before the present iteration works fine and is fully understood. Debugging and testing is an essential part of the process. Beginners will fail to find errors if the program is too complex and contains concepts they do not understand. So the starter project must be really simple and is probably not attractive in itself. Its beauty lies in the fact that it is the first step on the way to something interesting.

3 Text mining

Generally speaking, text mining means making profit out of text documents that are publicly available. The text is considered as a resource that can be exploited in order to produces additional value. Text mining can be considered as a threat, when someone searches for telephone numbers, names or e-mail addresses and misuses this information. But there are many useful applications like searching for rhymes or traffic information. An important concept in text mining is regular expressions. Programming novices have to learn two things, a) the general idea of pattern matching and b) specific formal details of regular expressions (placeholders like the dot . or operators like + and *) . The general idea of pattern matching is used (in a naive way) in everyday live, when we identify things or find things and separate them from others. Metaphors for regular expressions are

- a sieve that separates certain objects from other objects
- a "grabbing-device" that can only interact with objects that have certain surface properties (lock-key concept)

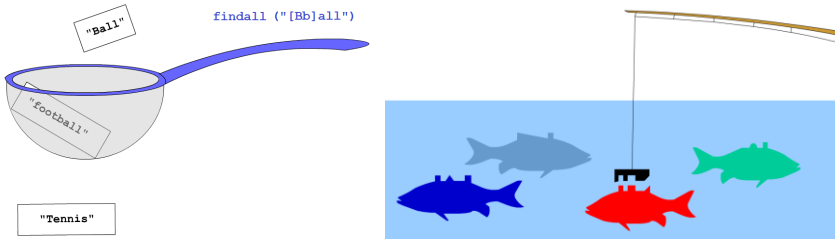

Fig. 1. Two different metaphors illustrating the concept of finding strings with regular expressions.

The formal details of regular expressions are best understood by reading the language reference and experimenting with individual statements. The function `findall()` from the Python module `re` takes a regular expression and a string as arguments and returns a list of all matching substrings. Here is a mini series of experiments illustrating how to find words that end with `"eep"`:

```
>>> text= "Keep it. Reeperbahn is a street in Hamburg."
>>> findall("\w*eep", text)
['Keep', 'Reep']
>>> findall("\w*eep ", text)
['Keep ']
```

3.1 Mining Mark Twain - Using Literature for Finding Rhymes

In the Project Gutenberg you can find 50 000 free e-books, including the entire works of Mark Twain (http://www.gutenberg.org/ebooks/3200). Download the utf-8 text file (15.3 MB) and store it in your project folder. This book (with 5598 pages)

can be used for finding rhymes. The following listing shows a starter program (Python). The call of `findall()` in line `#1` returns a sequence of words that end with the giving ending (plus a space). Statement `#2` transforms the list to a set (without duplicates) and prints it on screen.

```
from re import *
f = open("marktwain.txt", mode="r", encoding="utf-8")
book = f.read()
f.close()
ending = input('Ending: ')
while ending:
    wordlist = findall("\w*" + ending + " ", book)  #1
    print(set(wordlist))                            #2
    ending = input('Ending: ')
```

This is the output from an example run:

```
Ending: eep
{'sheep ', 'Weep ', 'Sheep ', 'deep ', 'asleep ', 'keep
', 'steep ', 'creep '. ...}
```

This program works nicely, but it has many obvious weaknesses. For example, the output could be prettier (no curly brackets, commas etc), capitalized duplicates should be eliminated (just `weep` instead of `Weep` and `weep`), and the space-symbols at the end of each word could be cut off. Learners can extend the starter project and implement more stories in further iterations.

3.2 Mining social media

Small programs are not per se easy to understand just because they are small. Some statements may adopt advanced programming concepts that are difficult to understand. Let me discuss an example.

The Python module `tweepy` supports accessing twitter tweets. When someone submits a tweet, this event is documented in a json-string that is publicly available. This record contains the text of the tweet as well as information about the tweeter. If you want to create an application for processing tweets, you need to register your application on the twitter website. You get some keywords, which your program needs for authentication (consumer key, consumer secret, access token, access secret). The following program implements this story (1): Select all tweets about “gaming” and “smart city” from a live stream and store them in a text file.

It runs until it is stopped by a keyboard interrupt. Within a few hours one can collect thousands of tweets, which can be analyzed later.

```
from tweepy import OAuthHandler
from tweepy.streaming import StreamListener
from tweepy import Stream

f = open('my_tweets.txt', 'w')
class MyListener(StreamListener):                   #1
```

```
    def on_data(self, data):
        f.write(data)
        f.flush()
        return True

auth = OAuthHandler('consumer key', 'consumer secret')
auth.set_access_token('access token', 'access secret')
listener = MyListener()
stream = Stream(auth, listener)
stream.filter(track=['smart city', 'gaming'])        #2
```

This is an object oriented program. It contains several object instantiations and the definition of a derived class (#1). The programmer must override the method `on_data()`, which processes each tweet that is taken from the Firehose. The parameter `track` defines a selection pattern. Twitter allows at most 1% of all tweets in the Firehose to be selected. Obviously, rather advanced programming concepts are involved. How to explain this to a beginner, who is diving into this technology?

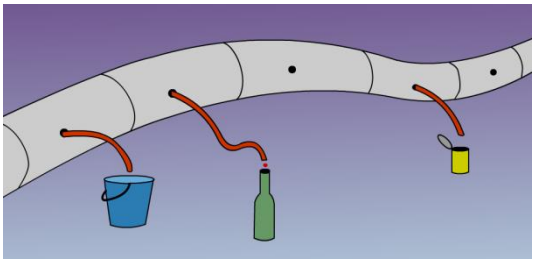

Fig. 2. Mining the Twitter Firehose.

Figure 3 gives an intuitive model of the whole project. In Extreme programming this is called a "project metaphor". It is one holistic idea how to mine a Twitter live stream. In addition one can map elements of the image to formal constructs in the program text: The Stream-object is represented by a big pipe, the AuthHandler-object – resposible for access to the Firehose – is visualized by a red pipe, the file storing tweets, is a container (bottle, bucket or can) and so on.

The text file containing collected tweets can easily be analysed using the standard methods of string objects. Example (Python):

```
>>> text = "This is a tweet."
>>> text.count("is")
2
```

Further stories could include these: (2) Check the frequency of tweets about topics like "gaming" or "smart city". (3) Estimate the average age of persons tweeting about certain topics by analysing the language they use.

An approach to implement story 3 is searching for certain stylistic elements that are age-dependent. For example, young tweeters use more often the words "I", "me", "you", capitalized words like "LOL" or "HAHA" and they use more often alphabetic

lengthening like “niiiice” instead of “nice”. Tweets from older users on the other hand contain more hyperlinks and references to the family (“family”, “son”, “daughter”)[12]. Table 1 shows the results of a “toy analysis” of tweets, which were collected during the same time slot (14 hours on May 17th 2015).

Table 1. Results from a toy analysis of tweets containing CS-related phrases.

Phrase	Number of tweets	Average length (words)	“Old” stylistic elements	“Young” stylistic elements
Smart city	226	24.6	2.8 %	7.4%
Internet of things	1718	27.9	4.4%	11.2%
Gaming	19928	26.4	3.6%	11,6%

4 Web cam analysis

Webcams make live at certain places really public. In contrast to surveillance cams which are accessible only by authorized persons, the images of public webcams can be observed and analysed by everyone. Running webcam-related programs implies interacting with the social environment. The input device is a public spot. This might provoke thinking about legal, political and ethical aspects of public webcams and digital technology (personal rights, security, and privacy).

Figure 3 shows screenshots from two different Python programs displaying and evaluating images public webcams.

Fig. 3: Screenshots from Python programs, showing and processing the live image of public webcams at the Friedensplatz in Dortmund, Germany (left hand side) and at a freeway junction at Frankfurt, Germany (right hand side).

The first application observes two areas (marked by white rectangles in the lower right quadrant of the image) and detects any motion at these spots by comparing the present picture with a photo taken a few seconds earlier. The application uses the marked areas for picking the answers of two teams in a quiz. Imagine questions with

two response options (yes and no). Each team answers “yes” by moving on “it’s spot”. It selects the answer “no” by keeping the area free from any activity. The second application observes a small rectangular area on a freeway, counts the number of motions in ten minutes, and estimates the density of the traffic [13].

Both programs consist of approx. 60 lines of code. A student – say Anna – could just copy such program from a text book. But if Anna is not familiar with the concepts included, this would not necessarily lead to comprehension. An alternative to copying letter by letter is *reconstruction*. Anna starts with a very simple nucleus, tests it until it is fully understood and then extends and changes it in iterations (similar as in Extreme Programming). This can be supported by the text book. Here is an example of a program which could be the first reconstruction step in both projects. Story 1: Get an image from a webcam and show it on screen.

```
import io
from urllib.request import urlopen
from PIL import Image
URL = "http://.../friedensplatz/current.jpg"
f = urlopen(URL)
imgText = f.read()
f.close()
imageBin = io.BytesIO(imgText)
img = Image.open(imageBin)                          #1
img.show()
```

This linear program just demonstrates how to get an image from the internet on the display of the computer at home. The image data must be transformed in several steps. Finally (in line #1) a PIL.Image-Object has been created.
Story 2: Draw something on the image, say a rectangle. This story is implemented by adding a few lines of code:

```
from PIL import ImageDraw
...
A = (305, 375, 325, 395)
...
draw = ImageDraw(img)                               #2
draw.rectangle(A, outline="white")                  #3
```

These few lines of code demonstrate the idea of a PIL.ImageDraw-Object. In line #2 a new ImageDraw-object (named `draw`) is created and connected to a PIL.Image-object named `img`. When `draw` receives a message (like in #3) it changes the state of the connected image. Further stories (which can be used in both projects) are: 3) Show the image in an application window and update it every *x* seconds. 4) Detect motion in two rectangular areas. 5) Show the results of the motion detection on a label below the photo. At some point refactoring is necessary. This means to improve the technical quality of the program (without changing its functionality). The target program is a well readable well structured object oriented program. Students will take the given program as an inspiration and add their own ideas.

Conclusion

Making CS relevant for young people is a major challenge for teachers and text book authors. Digital technology is omnipresent in our lives. But this does not guarantee that young people are interested in taking CS classes at schools. Fundamental principles and practices of CS must be imbedded in contexts that inspire young people. We need interesting project ideas (stories) that can be implemented quickly in small programs and media (images) that explain the idea of program code very quickly.

References

1. Seehorn, D., Carey, S., Fuschetto, B., Lee, I., Moix, D., O'Grady-Cuniff, D., ... & Verno, A.: CSTA K-12 Computer Science Standards, available at http://csta.acm.org/Curriculum/sub/CurrFiles/CSTA_K-12_CSS.pdf (2011).
2. Wing, J. M.: Computational thinking. Communications of the ACM, 49(3), 33—35 (2006)
3. Prensky, M.: Programming is the new literacy. Edutopia magazine (2008)
4. Gander, W.: Informatics and General Education. In Informatics in Schools. Teaching and Learning Perspectives (pp. 1-7). Springer International Publishing (2014)
5. Mayer, R. E., & Wittrock, M. C.: Problem solving. Handbook of educational psychology, 2, 287-303 (2006)
6. Ginat, D., Shifroni, E., & Menashe, E.: Transfer, cognitive load, and program design difficulties. In Informatics in Schools. Contributing to 21st Century Education (pp. 165-176). Springer Berlin Heidelberg (2011)
7. Sjøberg, S., & Schreiner, C.: The ROSE project: An overview and key findings. Oslo: University of Oslo (2010)
8. Papert, S.: Mindstorms: Children, computers, and powerful ideas. Basic Books, Inc. (1980)
9. Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... & Kafai, Y.. Scratch: programming for all. Communications of the ACM, 52(11), 60—67 (2009)
10. Lakoff, G., Núñez, R.: The Metaphorical Structure of Mathematics: Sketching Out Cognitive Foundations for a Mind-Based Mathematics. In L. English (Ed.), Mathematical Reasoning: Analogies, Metaphors, and Images (pp. 21-89). Hillsdale, NJ: Erlbaum (1997).
11. Beck, K.: Extreme programming explained: embrace change. Addison-Wesley Professional (2000)
12. Nguyen, D., Gravel, R., Trieschnigg, D., & Meder, T.: " How Old Do You Think I Am?"; A Study of Language and Age in Twitter. In Proceedings of the Seventh International AAAI Conference on Weblogs and Social Media. AAAI Press (2013)
13. Weigend, M. : Raspberry Pi programmieren mit Python, 2nd edition. MITP (2015)