Investigating the Usefulness of a Meta-Schema for
Program Design in an Introductory Programming Course

Miriam Brielmann

Thesis Submitted for the MSc in Mathematics and Science Education
Supervised by Dr. Wolter Kaper and Ben Bruidegom

AMSTEL Institute
Universiteit van Amsterdam
Science Park 404
1098 XH Amsterdam
The Netherlands

March 13, 2009
Abstract

This thesis describes a research project that investigates the effects of fostering a meta-schema for program design for novice programmers. The research builds on previous research which showed that the main difficulty faced in learning to program is related to planning rather than language structures. It further draws on a study of breakdowns of professional software designers which indicated that a meta-schema for design is helpful in the design process when the developer does not have a specialized design schema available from previous experience. A meta-schema for design is a schema guiding the design process, whereas specialized design schemas are applied for solving specific classes of problems.

We conducted the study in an introductory programming course with two groups of students of a Bachelor program in Computer Science at a university in Germany. In order to foster the development of a meta-schema for the students, the experimental group was specifically taught about stepwise refinement as a design strategy and structograms as a visualization of the design both as part of the units about algorithm visualization and software design. The control group only got an introduction to structograms as part of the conventional teaching of algorithm visualization. Another part of the intervention was the “sports club assignment” which was the lab assignment following the teaching about software quality and for the students the first opportunity to practice designing a program. The assignment instructed the students to plan the program, implement it and document the process. By giving more specific instructions on the planning, we again put more emphasis on the design for the experimental group.

Finally, we analyzed the resulting program code and the documentation the students handed in. The results do not show influence of our intervention on the correctness of the programs. However, there is a significant difference in the maximum complexities of the program modules, which means that the intervention had a positive effect on the problem decomposition and hence the changeability of the programs. Moreover, our observations confirm that the students do not have problems with applying the language structures they have learned about, but that the major problem is how to use them effectively. We conclude that an ideal curriculum for teaching programming to novices would foster the development of both meta-schemas (design strategy and visualization) and specialized design schemas (best practices in applying data structures and control structures). A curriculum revision for implementing this concept would have to include contents of courses that in general are only taught after the introduction to programming. For further research we propose to extend the study by conducting long-term observations and including the analysis of more quality criteria as well as the students’ perceptions of the intervention and the influence of their background on the effects. Also it seems desirable to investigate different ways how the meta-schema fostering activities can be integrated into a course, the role of the structograms as cognitive tools and the influence of the language style used in planning on correctness.
Acknowledgements

This work would not have been possible without the help and support of many people. In particular, I would like to thank my supervisors Dr. Wolter Kaper and Ben Bruidegom for their steady guidance, invaluable advice and endless patience throughout this endeavor.

Moreover, I would like to express my deep gratitude to Prof. Beate Bossler for giving me the opportunity to conduct the research in her classes as well as for her help and inspiration. I am also very grateful to the students for being amazingly nice and motivated research subjects.

I acknowledge Dr. Mary Beth Key, our program coordinator, for her continual support, and I sincerely thank all who have supported me in my studies and everyday life in Amsterdam: the staff of the AMSTEL Institute, especially Johan de Ridder, the International Office, fellow students, colleagues, room mates, and friends.

Special thanks go to Wolfgang Helbig for triggering the idea for this project and my former colleagues for standing our discussions during lunch breaks. I am indebted to Dorothea Koschmieder for being a wonderful host.

Finally, I cannot express how much I have come to appreciate my faithful friends and family who have encouraged and helped me all along the way. Thank you!
## Contents

1 Introduction ............................................................... 1

2 Theoretical Framework .................................................. 2
   2.1 Problems with Programming ........................................ 2
   2.2 A Meta-Schema for Design ......................................... 3
   2.3 Components of a Meta-Schema for Design ......................... 4
      2.3.1 The design strategy ........................................... 4
      2.3.2 Visualization of the program design ......................... 5
      2.3.3 Reasoning about program correctness and flexibility ....... 5
   2.4 Assessing Software Quality ........................................ 6

3 Research Aims ........................................................... 9
   3.1 Hypothesis ........................................................... 9
   3.2 Research Questions ................................................. 9

4 Research Setup .......................................................... 10
   4.1 Setting .............................................................. 10
   4.2 Intervention ........................................................ 11
      4.2.1 The Sports Club Assignment .................................. 11
      4.2.2 Characteristics of Correct Solutions ....................... 12
   4.3 Data Collection ..................................................... 13

5 Data Analysis ............................................................ 14
   5.1 The Product ........................................................ 14
      5.1.1 Setting up the Framework .................................... 14
      5.1.2 The Quality-Carrying Properties ............................ 14
   5.2 The Process ........................................................ 20

6 Results ........................................................................ 23
   6.1 The Product .......................................................... 23
      6.1.1 Program .......................................................... 23
      6.1.2 Module ........................................................... 24
      6.1.3 Sequence ........................................................ 37
      6.1.4 Loop .............................................................. 37
      6.1.5 Selection ........................................................ 42
   6.2 The Process ............................................................ 42
      6.2.1 Documentation .................................................... 42
      6.2.2 Changes .......................................................... 45
      6.2.3 The Students’ Reflections ...................................... 46

7 Conclusions and Discussion ............................................. 49
   7.1 Conclusions .......................................................... 49
   7.2 Discussion .............................................................. 51
      7.2.1 Evaluation of the research setup .............................. 51
      7.2.2 The findings ...................................................... 51
      7.2.3 Further research .................................................. 55
1 Introduction

The idea for this project arose from a certain dissatisfaction of the researcher with the quality of software, both as a user in everyday life and as a member of a professional software development team.

Examples of Problems Encountered

Let us look at an example from the work as an application developer for a radio station. As a new feature of our software for planning broadcasts, we offered a web-service that returned an overview of program data, so the schedule for a week could be easily presented on the Web. The service had been tested and deployed on time. Our clients started to use it and everything seemed to work fine.

Some time later however, suddenly there was an overload on the server and it was hardly possible to work with the software at all. Soon, the new web-service was identified as the cause. (It was the end of the year and the person who had programmed the methods for the web-service was on vacation!) While several people were busy taking calls from different important people from the client side, assuring them that this issue was handled with highest priority, another programmer finally found the bug in a simple loop. This loop that collected the data for a number of days did not reach its termination condition and looped infinitely when it started calculating the following year.

With our web-based tool for user administration we had another problem. It started off as a by-product, but became increasingly important as more people started to use the software. With that, more requirements on it were raised. However, for many reasons, no one really wanted to change it further. One was that the design was so convoluted that we had lost overview of what consequences a change in one part could have in another.

In a training session with a group of future user administrators, we received confirmation that the tool was not able to deal with many people using it at once. For example a click by one would change the data another had selected. Nevertheless, this was never properly fixed, because it was estimated that the probability that something would happen was low and the change would be too expensive. Luckily, nothing severely bad happened thereafter. However, user administration stayed a job nobody really wanted to do because of the unpredictable behavior of the software.

As these examples show, running into bugs can be very annoying and stressful. The importance of software tests must not be underestimated, but even after having found the cause of trouble during a test, one will often think that the error could have been avoided in the first place.

From an educator’s view, we are now asking the question: How can we, when teaching students to program, enable them to develop correct and flexible software?
2 Theoretical Framework

2.1 Problems with Programming

Experience tells that there are different factors that increase the probability of making mistakes in programming. Deadlines and time pressure often make developers forget or ignore what they have learned: that a thorough design of software will save time in the end. However, most probably customers often only figure out what they really want when the software is (almost) finished. These changes of requirements can lead to major redesigns of the software that must be very well planned and implemented. In cases like these, lack of experience and problems with basic planning and design also come to light.

Robins, Rountree, & Rountree (2003) review literature on what research has been done so far on the topic of learning and teaching programming. They describe “five overlapping domains and potential sources of difficulty that must be mastered” when learning to program:

1. General orientation, what programs are for and what can be done with them;
2. The notional machine, a model of the computer as it relates to executing programs;
3. Notation, the syntax and semantics of a particular programming language;
4. Structures, that is schemas/plans [...] ;
5. Pragmatics, that is, the skills of planning, developing, testing, debugging, and so on.

(Robins et al., 2003, referring to du Boulay, emphasis by the authors)

According to them, “writing a program involves” the following “mental models”:

1. “an understanding/mental model of (the) problem domain must precede any attempt to write an appropriate program”;
2. “…the central role played by a model of (an abstraction of) the computer, often called a ‘notional machine’”;
3. “The programmer must also develop a design/model of the program itself and how it will run”;
4. a “model of the program as it actually is”.

(Robins et al., 2003, emphasis added)

It is pointed out that “basic program planning rather than specific language features is the main source of difficulty” (Robins et al., 2003). The authors recommend that “instruction should address the underlying issue of basic program design, in particular the use of the schemas/plans which are the central feature of programming knowledge representation.” (Robins et al., 2003).

These conclusions are strengthened by the findings of Guindon, Krasner, & Curtis (1987). In their study about problems in software development, programmers with different kinds of background knowledge were observed as they were making a program design for “a problem of realistic complexity”. The origins of the breakdowns are categorized into three classes:
2.2 A Meta-Schema for Design

In Guindon et al. (1987)’s analysis of the breakdowns, the concept of “schemas” plays an important role. According to McCracken (2004),

“a schema is an abstraction (similar to an object in OO design) that can be instantiated by assigning values to its variables. Schemas can be realized as computational models or can be used as a non-computational model of memory. […] Expert designers have many schemas learned from experience, and extensive pattern matching and analogy generating processes that support problem solving” (emphasis added).

Guindon et al. (1987) identify two types of schemas that are relevant in the design process:

a) A specialized design schema is a design schema that is taken from the same domain as the problem to be solved. Guindon et al.’s research subject P6\(^1\) for example was experienced in designing distributed systems. He could make use of the knowledge he had gathered so far in this domain and that fit the pattern of the given task. He “use[d] these design schemas to decompose the problem into simpler subproblems.”

b) A meta-schema for design is more general. “A design meta-schema is a schema about the process of design itself and not about a particular class of problems. The meta-schema is used to control the design process. A design meta-schema guides the execution of design activities and resource management. A design meta-schema represents design process goals and their alternatives and guides the amount of effort spent in different activities.” P8 did not have experience in the problem domain, but could fall back on his experience in software design in general. That helped him to structure the process of design.

The third research subject described (P3) performs poorly in comparison with P6 and P8:

“P3’s design activities appear the least systematic and the most locally governed. They are the least systematic because P3 does not seem to use a meta-schema for design and he does not seem to use specialized design schemas to guide the decomposition of the problem into subproblems. […] His approach seems mostly governed by a familiar

---

\(^1\) Guindon et al. number their research subjects P1-P8 and describe the behavior of the three participants P3, P6 and P8.
computational paradigm, logic programming. He produces many cycles of generating a tentative solution, simulating it, debugging it locally, and simulating it again. He has difficulty with mental simulations of the many tentative solutions and with keeping track of the test cases during these simulations."

Looking at the three research subjects, the meta-schemas seem to be very helpful when developers are confronted with new types of problems, for example when someone is not an expert in a field (yet) and cannot retrieve specialized design schemas. A lack of design schemas also surfaces with the breakdowns of class three (breakdowns due to lack of knowledge and due to cognitive limitations), i.e. in the “difficulty in keeping track and returning to aspects of problems whose solution refinements have been postponed” and the “difficulty in expanding or merging partial solutions into a complete solution” (Guindon et al., 1987).

The hypothesis of Guindon et al. (1987) is that “the use of a meta-schema for design is particularly useful if the designer lacks more specialized relevant design schemas. The use of a meta-schema about design helps the designer control the amount of effort spent in different activities during design (e.g., explorations for understanding the problem environment, exploration and evaluation of different solutions)."

### 2.3 Components of a Meta-Schema for Design

For this research project, we take up the hypothesis of Guindon et al. (1987) and investigate the meaning of a meta-schema of the design process for novice programmers. However, while Guindon et al. (1987) stress the meaning of the timing and prioritization aspect of the design process we look at “meta-schemas” as everything that is not problem or domain specific. We are investigating the impact of the following three “components” of meta-schemas in particular.

#### 2.3.1 The design strategy

The idea of Dijkstra (1970)’s design strategy “step-wise program composition” (later called “step-wise refinement”, also often called “top-down design”) is to “shorten the conceptual gap between the static program text […] and the corresponding computations” (Dijkstra, 1970) in order to support the understanding of a program.

The strategy is to compose a program step by step, while each step is a refinement of a part of an earlier step. On every refinement level it is expected that the program is executed by a machine that knows the a very high level, problem specific instruction set, so the whole problem can be solved in just a few steps. In each refinement step, one of the “magical” instructions is translated into a lower level “language”, aiming to end at the programming language after a finite number of steps. Every step is as small as possible, i.e. as few decisions as possible are taken at a time, while the other decisions are postponed. The program can be verified as being (or clearly is) correct in every design step.
Moreover, the definitions of the instruction sets can be put in a row, just like the pearls in a necklace, as Dijkstra (1970) calls it. Each of the pearls represents a unit of the program solution and can be replaced by alternative solutions without affecting other parts of the necklace.

Another explanation of the stepwise refinement as well as examples can be found in appendix A.1.

I expect this strategy to enable students to write more flexible programs and to be able to react to changing constraints, because

1. the process is based on the thought that every single step is clear and easy to understand, so the resulting code should be easy to understand, too. By that, it should be easy to do changes for both the original developer himself and persons in charge later;

2. the use of “pearls” leads to modularity. By that, parts of the program can easily be replaced by new parts that are adjusted to the new requirements.

2.3.2 Visualization of the program design

Only a few years after Dijkstra’s notes on structured programming and his discussion of the stepwise refinement, Nassi and Shneiderman “propose[d] a flowchart language whose control structure is closer to that of languages amenable to structured programming” than the “conventional flowchart language” (Nassi & Shneiderman, 1973).

“Nassi-Shneiderman diagrams”, also called “Structograms”, consist of rectangular flowchart symbols that “represent the basic control operations that are available to the programmer”, combined to form structures.

The use of rectangular shapes and “the absence of any representation for the branch instruction forces the user to design programs in a structured manner” (Nassi & Shneiderman, 1973).

Noteworthy also is that for every box (= a process), “an entire structure [= a set of boxes] could be put in its place” (Nassi & Shneiderman, 1973). By that, the notation supports Dijkstra’s levels of refinement.

Unfortunately, due to the compact form of the diagrams, it is often rather cumbersome to put down design ideas on paper and work on them. According to our experience so far, students are demotivated by that, seeing only the disadvantages and not recognizing the use of the diagrams.

However, we expect them to profit from using the structograms because

a) they are a tool for writing down the levels of refinement and keeping track of them;

b) the structured visualization of the program design can be the subject of discussion and reflection.

2.3.3 Reasoning about program correctness and flexibility

We described that we expect that knowing and using a certain strategy and a way to visualize the design will decrease the probability of making mistakes.
However, taking the structogram and going straight to coding would still leave room for trial and error. At the latest when the program does not work, it is still tempting to end up in fiddling with the code and losing the overview of the whole.

As discussed earlier, checking the design for correctness before putting it into code can save a lot of trouble in the end. The same is valid for the flexibility of a program. Considering certain points before coding can save time later.

Our experience is that reasoning about a design, especially discussing it with others, helps one to make conscious decisions before it is necessary to re-design (or debug) in reaction to a problem. Thinking aloud in this way gives a certain security that the software delivered works. Furthermore, it helps to get a clear orientation of the structure of the program. So if there still is a problem, one can go back to the design and look for it there rather than search for the bug by trial and error in the program code.

Dijkstra is more precise and describes three “mental aids available to understand a program (or a proof of its correctness)”:  

1. *Enumeration*, i.e. going through the code step-by-step,

2. *Mathematical Induction*, i.e. verifying that a loop or recursion behaves as expected using a proof by induction,

3. *Abstraction*, e.g. using variables and “naming an operation and using it on account of ‘what it does’ while completely disregarding ‘how it works’” (Dijkstra 1970).

By practicing the skill of reasoning about their design, we expect that the students will learn to make conscious decisions in the design process rather than working by trial and error. Without explicitly teaching Dijkstra’s three principles, even in a first year programming course students could become sensible to the need for correctness checks.

2.4 Assessing Software Quality

The definition of “quality” has been subject to philosophical discussion. It is no surprise that there are different opinions about “software quality” and how to measure it. Over the years, a number of frameworks for measuring software quality have been developed. Amongst the best known examples are McCall, Boehm, ISO 9126 and Dromey (Ortega et al. 2003).

**Internal vs. External Quality Characteristics** In general, a distinction of quality attributes can be made based on what these attributes describe and how they can be measured:

- Internal quality characteristics are “relating to how the product was developed […], for instance: size, tests, failure rate, exchange rate, structure, etc. taken in the development of the product” (Ortega et al. 2003). They “can be measured by examining the product […] on its own, separate from its behavior” (Fenton & Pfleeger 1997).

- External quality characteristics “govern how the product works in its environment […], for instance, Usability and Reliability” (Ortega et al. 2003). They “can be measured only
with respect to how the product […] relates to its environment. Here, the behavior of the process, product or resource is more important than the entity itself.” (Fenton & Pfleeger, 1997)

Often, internal attributes are used as predictors of external attributes (Fenton & Pfleeger, 1997).

Out of the software quality models mentioned above, McCall’s and Boehm’s models are the oldest, created in the end of the 1970’s. Common to them is the compositional approach: They identify a set of external key attributes of the final software product (“quality factors”) that are relevant to the user, e.g. “reliability”, “usability”, and “maintainability”. Because these factors are considered to be at too high a level, they are further decomposed into “quality criteria” (e.g. with “correctness” being composed of “traceability”, “completeness”, and “consistency”). These quality criteria sometimes are internal attributes. The criteria are again broken down into “quality metrics” which are attributes that are directly measurable (e.g. “effort” and “change counts” for “expandability”).

In the early 1990’s, the ISO 9126 standard which was developed on the base of McCall’s model was proposed as an international standard for software quality measurement. There, “software quality is defined to be: ‘The totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs.’ ” (Fenton & Pfleeger, 1997).

The standard uses six quality factors which are decomposed into a number of sub-characteristics:

- **Functionality**: suitability, accuracy, interoperability, compliance, security
- **Reliability**: maturity, fault tolerance, recoverability,
- **Usability**: understandability, learnability, operability
- **Efficiency**: time behavior, resource behavior
- **Maintainability**: analyzability, changeability, stability, testability
- **Portability**: adaptability, installability, conformance, replacability

(Ortega et al., 2003). The sub-characteristics are not part of the standard but only given as examples in the annex. In opposite to the McCall and Boehm approaches, the metrics for measuring the characteristics are not part of the model (Ortega et al., 2003). Criticisms of the standard are that, due to its broadness, it “does not provide a conceptual framework within which comparable measurements may be made by different parties with different views of software quality, such as users, vendors, and regulatory agencies. Moreover, the definitions of well-studied attributes such as ‘reliability’ differ from other, well-established standards” (Fenton & Pfleeger, 1997). Nevertheless, the ISO 9126 model is widely recognized as standard and referenced as base for other software quality models (Dromey 1995, Sharma et al., 2008, Kalaimagal & Srinivasan, 2008, Chua & Dyson, 2004).

**Dromey’s Model for Software Product Quality** is connecting to the high-level attributes of the ISO 9126 Model. Dromey’s aim is not only to support the evaluation of some product’s quality, but also to facilitate “building quality into software” which according to him is a feature that is missing in the previous quality models. Central to Dromey’s model is a set of “quality carrying properties”. These are internal characteristics which are directly connected to and identifiable in the product’s components (“structural forms”) and related to the high-level quality attributes.
For applying the model, two perspectives complement each other: For implementation, quality can be built into the software by making sure that for each structural form the applicable quality carrying properties are fulfilled (bottom-up, see figure 1). For design, it is useful to take the high-level quality-attributes as starting point and identify the relevant quality-carrying properties (top-down) [Dromey 1995].
3 Research Aims

3.1 Hypothesis

Concluding from the literature review, we state as hypothesis to be evaluated by the research project:

\[\text{Fostering the building of a suitable meta-schema for program design will support students in learning to develop correct and flexible programs.}\]

For the evaluation of the given hypothesis we focused on the following components of meta-schemas:

1. Design strategy,
   Represented by the “stepwise refinement” method;

2. Visualization of the program design,
   Represented by Nassi Schneidermann diagrams (Structograms).

We also expect reasoning about program correctness and flexibility to happen as part of the activities. However, we did not devise any special method for it and expect just the use of “common sense”. This choice was influenced by the choice of our research setup: a first year undergraduate programming course (see section 4).

In our literature search, no previous attempt to teach this specific meta-schema in an introductory programming course has been found.

3.2 Research Questions

Proceeding from the components of meta-schemas given above, we pose the following research questions:

1. How does the teaching of “stepwise refinement” as a design strategy enable students\(^2\) to decompose a problem and react to changing constraints?

2. How does the use of documentation and visualization (writing down stepwise refinement, using structograms) support the process of learning to design software?

3. Do stepwise refinement and visualization influence the quality of the resulting code?

Because a good decomposition, parametrization and changeability are quality criteria, question 1 is really a part of question 3. However, question 1 asks straightly for the influence of stepwise refinement on those quality criteria that we expect to be influenced the most (see the hypothesis), while question 3 asks about the effect on quality in general.

\(^2\) of a first year undergraduate programming course
### 4 Research Setup

For our investigations about the given research questions, we set up an intervention that emphasized the use of stepwise refinement and structograms in a first year university programming course and analyzed the results.

#### 4.1 Setting

The research project was conducted at the University of Cooperative Education Stuttgart (Germany) in cooperation with Prof. Beate Bossler. The main part of the project was an intervention in a C-programming course for first semester undergraduate students. The research involved an experimental group of 25 students of applied informatics and a control group of 18 students of engineering. The students had been assigned to the groups beforehand, according to the focus of their studies. The researcher was involved in the programming course as teacher for the intervention and as a lab assistant for a number of lab sessions.
4.2 Intervention

The intervention consisted of a set of activities embedded in the curriculum of the course as shown in table 1. As can be seen in table 2, the activities differed between the experimental and the control group. The experimental group was taught about stepwise refinement which was introduced together with structograms. Both of the strategies were followed up on in the later unit about software quality. The control group did not learn about stepwise refinement and were asked to use structograms in the respective unit only.

After the introduction to software quality follows a lab exercise which is the students’ first chance of doing some larger scale planning (writing more than one function and deciding what to include in each of them). We chose this exercise as the main site of our intervention and prepared two different versions of the exercise for the experimental and the control group. This activity serves a dual purpose: on the one hand it is part of the learning phase. The experimental group students need to practice the learned theory in a first realistic case. The control students need to practice the spontaneous planning strategies that they also would have practiced without our intervention. On the other hand, this lab activity also has turned out as the main test of the effectiveness of the intervention. So, learning phase and testing are mixed here. The pros and cons of that setup will be discussed after the results are presented (see section 7.2).

In the following, we give a summary of the main activity, the sports club assignment as well as the characteristics of correct solutions. For a complete example design and implementation, please refer to appendix C.

4.2.1 The Sports Club Assignment

The main activity that we observed was conducted after an introduction to using arrays and a unit about software quality. You can find the full assignment in appendix B. The task puts the student into the context of working for a programming agency and being contracted by a sports club for programming a tool that prints a statistics table of the activities of the members of the club. The assignment consists of three parts of which the students only get one to work on in the beginning. However, with the first part it is announced that more requirements will follow. These are handed out as part two and three after the students have been working on part one for about an hour (or when they finish working on part one). In addition to the programming problems, the students are given the task to design and document the program in advance, and to reflect on it in the end.

The programming task is the same for both of the groups. However, we differed in the questions we asked the students about the design process. For the experimental group instructions are given how to proceed for the design and what to document, including the instructions to apply structograms and stepwise refinement. For the control group the task description is short and general.
4.2.2 Characteristics of Correct Solutions

In the assignment we announce that changes in the requirements are to be expected. This is intended to motivate the students to keep the solution as general as possible. To achieve this, a student could realize that the following generalizations should be easy:

- The assignment talks of persons but this is not essential, any kind of items could be input and counted like it is described in the assignment text;
- The assignment talks about gender and sports type as two properties that people may have, but instead the assignment could just as well be about any two properties that the arbitrary items may have.
- It is essential that the number of properties is 2, and not more nor less, because we are required to print a 2-D table with one property laid out horizontally, the other one vertically. Therefore this number 2 will not be easily configurable, without getting a very different assignment.
- The two properties should be of a kind that take a finite set of string values (like gender: “male”/”female”, or sports type: “football”, “basketball”,...), and to be practical, one of the two sets of strings should fit when printed horizontally on an A4 page.
- For each of the two properties, the set of values should be numbered (0, 1, 2, ...) such that each value can be input by typing a number-code.
- For the output, the string values should be used as row-headers and column headers in the table to be printed.
- The body cells of the table should contain a count of the items having that particular combination of property-values (for instance a count of females doing basketball, or a count of books having red covers and being non-fiction)

However, though the assignment invites generalization, we did not expect students to arrive at such an explicit generalization as formulated above. Instead we were curious how far they would get in achieving extensibility and changeability on their own strength of reasoning. We will however use the terminology introduced above (items, properties, sets of values, integer codes to represent those values) to speak about the assignment in generalized terms, whenever necessary when discussing results.

Though students were asked to program as generally as possible, they were not asked to provide a user interface for configuring the program. We may expect, therefore, that any configurability will be achieved using a “configuration” section in the programs source code.

Various approaches to a solution will be considered correct. The main two different ones have to do with the data structure shared between input and output routine:

1. Inevitably there will be an input loop. Each time, two property-codes will be typed. These can be stored in a $P \times 2$ array for later processing and output ($P$ is the number of items/persons).
2. It is also possible to decide, seeing that all the items need to be counted once in one of the table-cells to be printed, to immediately set up a $M \times N$ array ($M$ and $N$ being the sizes of the value-sets for both properties), initialize all array cells to zero, and then during the input loop, increase the counter in the appropriate cell in each iteration of that loop.

Students choosing design (1) have one data structure extra to take care of, because they will need the $M \times N$ array later anyway. Still, design (1) must be considered better from the point of view of extensibility, because the stored pairs retain their identity and could later be used for other purposes.

Our ideal design would therefore consist of:

- Initialization: configuration statements to initialize constants like number of persons $P$, and the sizes of the two values sets $M$ and $N$. Possibly also: to initialize one $M$-sized and one $N$-sized array containing the string values for each of the two properties. However, string arrays were not treated yet, so only a few students would be able to do this.
- Inputting a maximum of $P$ items by asking 2 property-codes from the user each time and storing them in a $P \times 2$ array.
- Processing the $P \times 2$ array by iterating over it and increasing in each iteration one of the counts held in an $M \times N$ array (initialized with zeroes first).
- Calculating column-totals for each of the columns in the $M \times N$ array and storing them in an $N$-sized array (initialized with zeroes first).
- Outputting the table:
  - printing the column headers, being the string values of the horizontally printed property
  - repeatedly printing a row:
    * print header of row i, being the string value of the vertically printed property
    * print the body cells for row i, which is just row i of the $M \times N$ counts array
    * print the sum of the just-printed body cells as a row-total
  - print a line
  - print the column footers: the totals of the columns

A C-program that conforms to this design is shown in appendix C.1.2.

4.3 Data Collection

The students of both of the groups were asked to hand in the program code as well as the documentation and their notes for the main intervention (the sports club assignment). We used these documents for analyzing the final product and the process of software design.
5 Data Analysis

For setting up and testing our framework for data analysis, we do a pilot check on ten programs, five samples of the control group and five of the experimental group, and the documentation connected to them. The analysis of the program code serves to answer research question 1 about the problem decomposition and changeability of the program in specific as well as research question 3 about the software quality in general. The analysis of the documentation serves to answer research question 2 about the process.

5.1 The Product

We analyze the program code as the final product in order to find out if the intervention had some effect on the students’ ability to decompose a problem and to react on changing constraints (research question 1) and the quality of the resulting code (research question 3). In total, we analyze 68 files of 38 students who handed in at least one useful program. For each of the students we analyze at the most one program per part of the assignment. We filter out files that included less than two of the three main functionalities input, counting and output. If more than one file was handed in for one part of the assignment, we select the one with most progress.

5.1.1 Setting up the Framework

For the analysis of the software quality for research questions 1 and 3, we choose Dromey’s model for software product quality ([Dromey 1995](#)) described in section 2.4 as guideline, because it provides a set of concrete quality characteristics that can be identified in the structural forms of the given code. However, we reduce the scope of our analysis by focusing on both the higher-level structural forms and the quality-carrying properties that we expect to be influenced by the intervention. In our analysis we include the structural forms module, sequence, loop and selection (with loop and selection being sub-categories of “statement”). Because of that we can immediately filter out the following properties that only apply to lower-level structural forms: Computable, assigned, precise, variant, nonredundant, direct, adjustable, encapsulated. For a description of the properties and how we select them in more detail see below. Finally, we add the properties “cyclomatic complexity” and “depth of the call tree” to our framework because decomposition is of special interest to us and this aspect is missing in Dromey’s model. For that and our general program check, we introduce the program itself as a new “structural form”.

For answering research question 1 the properties complexity, depth of call tree and parameterization are particularly important as they describe the decomposition and changeability of a program. The complete set of properties is relevant for research question 3 about software quality in general.

5.1.2 The Quality-Carrying Properties

The reductions of Dromey’s model have effects on our tracking of quality defects. Dromey defines as precedence rules “to always associate quality defects with the lowest level structural form they apply to” and “to classify a problem as correctness problem before structural, before modularity,
### 5.1 The Product

<table>
<thead>
<tr>
<th>Structural Form</th>
<th>Quality-carrying property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Correct</td>
</tr>
<tr>
<td>Module</td>
<td>Max. complexity</td>
</tr>
<tr>
<td></td>
<td>Depth of call tree</td>
</tr>
<tr>
<td></td>
<td>Complete</td>
</tr>
<tr>
<td></td>
<td>Consistent</td>
</tr>
<tr>
<td></td>
<td>Utilized</td>
</tr>
<tr>
<td></td>
<td>Parameterized</td>
</tr>
<tr>
<td></td>
<td>Loosely coupled</td>
</tr>
<tr>
<td>Sequence</td>
<td>Structured</td>
</tr>
<tr>
<td></td>
<td>Cohesive</td>
</tr>
<tr>
<td>Loop</td>
<td>Complete</td>
</tr>
<tr>
<td></td>
<td>Initialized</td>
</tr>
<tr>
<td></td>
<td>Progressive</td>
</tr>
<tr>
<td></td>
<td>Consistent</td>
</tr>
<tr>
<td></td>
<td>Resolved</td>
</tr>
<tr>
<td></td>
<td>Effective</td>
</tr>
<tr>
<td>Selection</td>
<td>Complete</td>
</tr>
<tr>
<td></td>
<td>Consistent</td>
</tr>
<tr>
<td></td>
<td>Effective</td>
</tr>
</tbody>
</table>

Tab. 3: The set of the quality-carrying properties applied for our analysis
before descriptive problems” [Dromey 1995]. We diverge from that strategy and decide to note the quality defects on the level where they become visible. One example of how we adjust the strategy to our purposes is a counter variable that is modified in order to exit a loop and by that simulate a “break” statement. According to Dromey this would be an inconsistence on variable level because the variable is used for two purposes (counting and exiting the loop at a certain point). We note it as an inconsistence on loop level because this is the smallest structure needed to see the defect. (After all, for our research it is more important to note that this leads to the sequence not being structured.)

In this section, we describe our interpretations of the properties from the Dromey model relating to the higher-level structural forms and the decisions whether to take them into consideration, and include our additional properties. The complete list of the resulting quality-carrying properties is given in table 3.

Correctness Properties The first set of properties is related to correctness and describing the procedural properties of the program, i.e. what it is doing and how. A violation of any of these properties can lead to the program not fulfilling its intended functionality.

Before going into the detailed analysis, we checked the program for general correctness and tested whether it compiles without changes. We evaluated how many features were not implemented and how many of those that were implemented were wrong. We collect bugs on the program level, as the students were free to choose their own distinction of functions and on the lower levels only incompletenesses, i.e. when parts are missing, will be tracked as defects.

- **Complete** (“A structural form exhibits the property of being complete when it has all the necessary elements to define and implement the structural form so that it may fulfill its intended role in a way that will not impact reliability or functionality.” [Dromey 1995])
  The property applies to object, modules and statements. According to our interpretation, “complete” means that something fulfills all the requirements given in the specification. Not complete is everything that is less than complete (compare to “consistent”). However, while the given sports club assignments are a specification of the program in total, we did not have an extra specification of the functions as designed by the students. As strategy to check for a function’s completeness we choose to take the documentation (if available) and the name of the function as basis for estimating what the function is supposed to do. If a part seems to be missing, we check if this part is taken over by another part of the program. If this is not the case, we mark the function as incomplete. We count loops as “complete” if they are guaranteed to terminate.

- **Initialized** (“A loop structure is initialized if all variables in a loop are initialized prior to loop entry and as late as possible prior to loop entry.” [Dromey 1995]). The core of this sub property of the “complete” property is the identification of the loop invariant. We keep track of this item as it is a critical point in the design of a program. We check whether a loop invariant was identified and compare it with our expectations.

3 The properties selected for the analysis are marked with bold font.
5.1 The Product

- **Progressive** ("A loop or recursive algorithm is progressive if there is clear evidence that the structure makes progress towards termination with each iteration or recursive call and the associated variant function is bounded below by zero." Dromey, 1995) It applies to recursive modules and loops and is also a sub property of the “complete” property. Because the problem given in the assignment is in a form that encourages “standard” loops, we suppose there will be no non-trivial loops and there will not be problems with progression. However, we decide to note irregularities regarding this property.

- **Consistent** ("A structural form is consistent if its usage maintains its properties or functionality and if all its elements contribute to and reinforce its overall intent or effect." Dromey, 1995) This property applies to modules, statements, guards, expressions, variables and records. The consistency is coupled to documentation: according to our interpretation something is consistent if it does what the documentation says and nothing more. In particular, it does not have any side-effects. If something is not consistent, it is more than complete. We decide to call only those functions “not consistent” in which an additional feature seems to be used on purpose. Bugs are noted as wrong and marked with a “-.” In the “consistence” column:
  - In one example (E12, assignment 2, Einlesen), a function is not doing what was explained in the comment, taking another value for quitting the input. However, we do not mark this as inconsistence because the feature is required in the specification (the comment). Instead we mark it with a “-” and note it as wrong.
  - Similarly, in another example (E30, assignment 1) there is one loop too much, which is obviously a bug because too many members are counted. Therefore for us this is noted as wrong and marked with a “-.”

**Structural Properties** The next set of properties is related to structure, i.e. in what form the functionality of the program is implemented.

- **Structured** ("A structural form exhibits the property of being structured if it follows rules of structured programming. That is, there should be only a single point of entry and exit for every control structure." Dromey, 1995) It applies to sequences, guards, and expressions and means for example that there should be no breaks, continues and go to’s. We track it for the sequences as it is an important feature that we discussed with the students in the context of the introduction to the structograms.

- **Resolved** ("A structural form is resolved if the control structure of the implementation involved matches the structure of the data or the problem in the sense advocated by Jackson (that is, the control structure matches data structure and thereby satisfies the correspondence principle)." Dromey, 1995) The property applies to loops. An example defect is the “use of a single loop to process a two-dimensional array” (Dromey, 1995).

- **Homogeneous** ("An iterative or recursive form is homogeneous if it can be described by an invariant where the major predicates assume a conjunctive form (e.g., the invariant must
be of the form ‘A and B and . . . ’ but A etc. may involve disjunction.” [Dromey 1995] It applies to loops and recursive modules. In the pilot programs we did not find examples of forms that contain combinations of conjunctions and disjunctions. Because of that we do not check this property.

- **Effective** ("A structural form exhibits the property of being effective when it has all the necessary elements and only the necessary elements to define and implement the structural form." [Dromey 1995] It applies to expressions and statements. We track this property as the negation of it, i.e. if there is code that can be deleted without having effects on the program, it contains ineffective code. It is a form of over-completeness, containing code that is not helping to reach completeness (but also not making the part inconsistent).

- **Range-independent** ("A structural form is range-independent if both its lower and upper bounds are not fixed numeric or character constants." [Dromey 1995] This property applies to declarations (arrays) and loops; we cover it by our parameterization analysis. However, in our opinion the lower bound does not need to be flexible if it has the value 0 or 1.

- **Utilized** ("A structural form is utilized if it has been defined and then used within its scope." [Dromey 1995] Applies to objects, modules and all forms of declared data and expresses whether a function is used. We track it for the modules.

**Modularity Properties**

The modularity properties describe the design of modules and how they relate to each other.

- **Depth of call tree** The depth of the call tree is discussed by Fenton as one of the attributes of the morphology of the call graph. The call graph is a graph with the modules as nodes and arcs between the nodes that are calling another. As the metric is measuring the interaction of the modules, it is called an inter-modular measure. [Fenton & Pfleeger 1997] We count it as the levels of function calls, including the call of the main function.

- **Cyclomatic complexity** The cyclomatic complexity is an intra-modular measure, measuring the complexity of a module by the number of decision points and accordingly the number of independent paths. The metric was introduced by McCabe who advised to consider a redesign of all modules that have a cyclomatic complexity greater than 10 ("which seems like a reasonable, but not magical, upper limit" [McCabe 1976] in order to keep the code testable and maintainable. In practice, the measure is used as a rule of thumb in different ways ("For each module, either limit cyclomatic complexity to 10 [...] or provide a written explanation of why the limit was exceeded." [Watson et al. 1996] The original, usual limit for a maximum acceptable value for cyclomatic complexity is 10. Other values, such as 15 or 20, have also been suggested. Regardless of the exact limit, if cyclomatic complexity exceeds 20, you should consider it alarming.” [Aivosto Oy 2008] We adopt Aivosto’s idea and group our students’ implementations into three groups, taking the conservative limit of 10 as upper bound for low complexity, allowing complexities from 11 to 20 as moderate, and marking complexities above 20 as high.
We use the software “Understand”\textsuperscript{4} for automatically generating the cyclomatic complexities for the programs of the students. For deepening the understanding of the measurement described by Fenton & Pfleeger \cite{Fenton:1997}, as well as for being able to do the measurement on the structograms by hand and to decide which of the different variants that are offered by the “Understand” tool to use, we did a small study on how to measure the cyclomatic complexity intuitively in a simple way. For that we transformed the formula into a faster way of counting the decision points and verified that the result is the same as for the original formula. Finally we choose to measure the “ordinary” version that counts all the if- and case-statements, but does not take into consideration conjunctions and disjunctions in conditions.

- **Parameterized** ("A module is parameterized if it contains as parameters all and only the necessary and sufficient inputs and outputs to characterize a particular well-defined function/procedure." \cite{Dromey:1995}) It applies to modules and describes what is configurable. We note it on file level in order to be able to compare the programs as a whole. An analysis on module level would be difficult because the students were free to choose their own way of dividing the problem into functions. We identified the list of features which we hoped the students to foresee changes for and check how parameterizable the students’ are concerning these points.

- **Loosely coupled** ("A module or a program is loosely coupled if all module calls are data-coupled to the calling program/module." \cite{Dromey:1995}) That means that the behavior of a function is not controlled from outside. For better understanding, we use Fenton’s description of coupling relations \cite{Fenton:1997}:
  
  - no coupling (no communication, completely independent)
  - data coupling (communication by parameters not control elements)
  - stamp coupling (same record type as parameter)
  - control coupling (passing a parameter to control the behavior of the function)
  - common coupling (using the same global data/variables)
  - content coupling (branch into, change data or alter statement in another function)

According to Dromey only data coupled calls are loosely coupled. Fenton puts data coupling and stamp coupling into this group.

We decide that the passing of pointers as parameters counts as “data coupling” \footnote{“Understand” by Scientific Toolworks: \url{http://www.getunderstand.com/}}. Therefore, the functions are still loosely coupled. Another decision is that “loosely coupled” refers to the coupling of the called function to the calling function. E13, assignment 2, eingabe for example is loosely coupled because it is not coupled to the main function. However, the functions called by eingabe are connected by global variables (common coupling) and are not loosely coupled because of this.

In addition, we check for the number of parameters of each function. If a high number of pointers/variables are passed as arguments of the functions, we can consider it a form of...
Data Analysis

stamp coupling, because as the parameters of one function change, the parameters of the other function will most probably have to be changed, too.

- **Cohesive** (“A structural form is cohesive if all its elements are tightly bound to one another and they all contribute to achieving a single objective or function.” [Dromey 1995]) It applies to sequences. We make a qualitative statement about the function in general (Fenton & Pileeger 1997): Can the module’s purpose be described in one sentence? Yes/no.

- **Generic** (“A module is generic if its computations are abstracted to a type-parameterized form.” [Dromey 1995]) Applies to modules. We skip this, because it is about type dependence, like writing a swapping function specifically for integers (see Dromey, p. 153). Because in C all functions have to be implemented for a certain type this is not criterion is not applicable for us.

- **Abstract** (“An object/module is sufficiently abstract if there is no obvious, useful higher level concept that encompasses the structural form.” [Dromey 1995]) Applies to objects, that is why in a C course we must ignore it.

**Descriptive Properties** We do not include the descriptive properties because we could not classify the variations in the pilot programs. We gave priority to describing the quality of the code itself rather than the documentation.

**5.2 The Process**

**Documentation** In the assignments, we asked the students to document the planning of their programs and hand in the documentation. We analyze this information, consisting of text and/or diagrams, for finding out in how far this process helped them to design the software (research question 2).

We also use these items for finding out in how far the students applied the strategy of stepwise refinement. Our approach is visualized in figure 2. We check for indicators of the students’ strategies and give them points from 0 = no planning/no information to 5 = good strategy/conforming with the strategy taught the most. The checking steps are:

1. Is there any description or visualization of planning at all?

2. Is stepwise refinement recognizable in any form (structograms or planning of functions)?

3. Is the stepwise refinement done in structograms?

4. Can the refinement tree be drawn?

5. Is the complexity of the structograms low?

Moreover, we see whether the structograms are recognizable in the program code and note the strategy that the students used for refinement and categorized them. Finally, we summarize our rating of the students’ understanding of stepwise refinement.
5.2 The Process

Fig. 2: Our approach of analyzing the design strategy
Changes  As measurements of changes from the first part to the second part of the assignment, we count the number of locations where changes were made for each changed requirement and check the number of changed lines in the program code.

The Students’ Reflections  In addition to the programming assignments, we asked the students to reflect on the process and the changeability of their code in particular. We look for points where students realize that their initial design or original implementation was not good.
6 Results

Results of our intervention fall into two main categories: analyses regarding the product of students’ work (6.1), and analyses regarding their documentation of the design process (6.2), as described in section 5.

6.1 The Product

The following analysis of product quality follows the framework set up in section 5.1 and summarized in table 3. Results are given for each of the quality criteria, ordered by the structural from to which they apply: program (6.1.1), module (6.1.2), sequence (6.1.3), loop (6.1.4) and selection (6.1.5).

6.1.1 Program

Correctness. Our first check is whether the programs can be compiled and work as they are supposed to. A part of the bugs that are there only comes to show if invalid numbers are entered. Because we had given the students the freedom to expect valid input, we distinguish the bugs happening for that reason, and the ones that occur even for valid input: Both from the experimental and the control group, we find 7 programs by 5 students that contain bugs that are solely due to the missing input check. In Dromey’s terms we could track them as a defect of “specified” (the programs expect valid input which is not stated explicitly). Other than them, we have 16 programs by 12 students from the experimental group and 14 programs by 11 students from the control group that contain bugs independent from the expected input. In general, we can group most of the mistakes into the following two categories.

The probably most obvious error is when the counting of the members for the statistics goes wrong. In some cases the counting is not correct by itself because the array used for the input and/or counting is not initialized (1 from experimental group, 3 from the control group) or counters are not increased when they are supposed to be (2 from experimental group). In this category also the failures due to missing input check occur: the counting only does not work if invalid numbers are entered. Then, the bugs are caused by the missing input check (which was not an explicit requirement) combined with a counting functionality that expects the input values to be valid. One way how this wrong counting happens is in a selection that takes one of the limit values as the default (see for example listing 1). Another way is when the 2d counting array is also for totalizing each row/column by reserving a last column/row for that purpose (see listing 2). In addition to the “specified” defect named above, the use of the same array for two purposes would be declared as inconsistence on variable level (which we do not track) and the selections not covering all possible cases as incompleteness (see section 6.1.5 about selections).

Another category of bugs is the dealing with the number of members. A group of students is not using any upper limit for the number of members (3 from the experimental group, 3 from the control group) which is formally not a correctness issue but an incompleteness. We do not count it as a bug. Another group is using an upper limit, but setting it to another value than required in the assignment (6 students from the experimental group, 1 from the control group). This often
### Tab. 4: Overview of the frequency of implementations with a certain depth of the call tree per group

<table>
<thead>
<tr>
<th>Group</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXP</td>
<td>5</td>
<td>25</td>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>CON</td>
<td>15</td>
<td>13</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>Total</td>
<td>20</td>
<td>38</td>
<td>10</td>
<td>68</td>
</tr>
</tbody>
</table>

seems to be done deliberately to reach an earlier stop of the input loop. Some students are using an upper limit and applying it the wrong way, so that loops are exited too early or too late (2 from the experimental group, 3 from the control group).

Although we expected that the students in the experimental group would perform better as a consequence of the improved planning, this is not the case. Overall, the number and types of bugs are approximately equally spread between the experimental and the control group. There are 13 programs and 6 students from the experimental group and 11 programs and 3 students from the control group which we did not note any mistakes for. If we do not count the programs that fail if there is invalid input as incorrect, we can add 3 “successful” students in each group, resulting in 9 out of 21 students with working programs in the experimental group and 6 out of 17 students in the control group. In our opinion, both of the numbers are rather low. We conclude that in this task the students are actually very busy with grasping and implementing the basic functionality of the program. Neither of the groups gets to the point of paying enough attention to the details. The time pressure might have played a role in that.

### 6.1.2 Module

We analyze a list of 202 modules, 123 from the experimental group and 79 from the control group.

**Depth of Call Tree** Our expectation was that the students who were taught stepwise refinement would produce programs with functions of a deeper call tree.

Table 4 shows that most of the implementations’ call tree has a depth of two (38). About half of that number (20) have a call tree of depth one (20) and again half have a call tree of depth three (10). The majority of the “two”s can be explained by the way the assignment was formulated for the experimental group: “create structograms on at least two levels [...] , divide your design into functions [...].” This gets clear when we look at the numbers for the experimental group compared to the control group: As expected, most of “two”s come from the experimental group, while in the control group the depths of one and two are almost equally spread. In class, some of the students of the experimental group had mentioned that they did not think it was necessary to split the problem in different parts. The activity motivated the students in the experimental group to try out functions. It would be interesting to find out how many students of the experimental group only chose the two-level design because they had to and how many students of the control group chose the one-level design because they did not think of anything else.
for(i = 0; i<=MAXANZAHL; i++) {
    if(Liste[i][0] == 0) {
        if(Liste[i][1] == 0) {
            fm +=1;
        } else {
            fw +=1;
        }
    } else if(Liste[i][0] == 1) {
        if(Liste[i][1] == 0) {
            hm +=1;
        } else {
            hw +=1;
        }
    } else if(Liste[i][0] == 2) {
        if(Liste[i][1] == 0) {
            vm +=1;
        } else {
            vw +=1;
        }
    } else if(Liste[i][0] == 3) {
        if(Liste[i][1] == 0) {
            tm +=1;
        } else {
            tw +=1;
        }
    }
}

Listing 1: The counting goes wrong if input is invalid (case 1: unspecified/incomplete)
/* Funktion zur Eingabe der Daten, welche dann in das Feld gespeichert werden */
void Eingabe(int Daten[4][3])
{
    int i=0,Frage=0;
    int Sportart=0, geschlecht=0;
    while (i<MAXMITGLIED && Frage == 0)
    {
        printf("Fussball-0 Handball-1 Volleyball-2 Tennis-3\n");
        printf("Welche Sportart : "); scanf("%d",&Sportart);
        printf("m"annlich(0) oder weiblich(1) : "); scanf("%d",&geschlecht);
        printf("weiteres Mitglied?(ja-0 nein-1) "); scanf("%d",&Frage);
        Daten[Sportart][geschlecht]+= 1;
        Daten[Sportart][2]+=1; //counting the sum in any case!
        Fragen++; //counting the sum in any case!
    }
    return;
}

Listing 2: The counting goes wrong if input is invalid (case 2: unspecified/inconsistent)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Low (1-10)</th>
<th>Moderate (11-20)</th>
<th>High (&gt;20)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>12</td>
<td>7</td>
<td>38</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>31</td>
<td>24</td>
<td>13</td>
<td>68</td>
</tr>
</tbody>
</table>

Tab. 5: Overview of the maximum complexities per depth of call tree

**Complexity** For comparing the complexities of the programs, we selected the module with the highest complexity of each program.

Firstly, we expected that a deep call tree would lead to a low complexity of the functions. Relating the complexity to the **depth of the call tree** (see table 5), we see that the implementations with one big main function are mostly of moderate complexity. The distribution of the implementations that are divided into functions (i.e. depth of two or more) is biased to low complexity. There is no change of the distribution from depth two to depth three. There is even one implementation with depth three that is of high complexity. Our interpretation of the numbers is that splitting the problem into functions helped most of the students to extract pieces of lower complexity. However, not all of the students managed to split the problem in a way that produced small, or even equally small chunks. We can identify the following patterns: (1) Solving the problem in one big main function, (2) extracting some pieces of the program into functions, while leaving big parts of the functionality in the main function (see listing 3), (3) using functions,
int eingabe(int sportler[999][3]) {
    // explanation of input
    // read gender
    // read sports type
    // check for validity of input
    // return number of members
    // complexity: 5
}

void main(void)
{
    // initialize counting array
    // count member data and store in separate array
    // output
    // check if continue for another club
    // complexity: 23
}

Listing 3: An example for splitting out one task into a function, but still keeping the main complex (complexity of main is 23, of input 5)

but overloading the functions by putting more functionality into them than the name documents (see listing 4) 5, (4) Splitting the problem into functions of balanced size.

Next, for the complexity change between the parts of the program, we saw two possible options: (1) In the best case, the complexities would go down, because the students improved their programs, (2) Alternatively, the complexities would go up, because the students recognized weak points and restructured their programs. Looking at how the complexity changes between the parts of the assignment (see table 6), it turns out that in part one most of the implementations

5 Note that this does not need to result in a complexity problem, as the functionality can still be spread over a number of small functions. Formally, this design breaks the rule of consistence.

void main()
{
    eingabe();
    system("PAUSE");
}

Listing 4: An example for overloading a function, “eingabe” is not only dealing with the input but also with everything else
have a low (18) or moderate (14) complexity and only a few a high complexity (2). But, as we were afraid, in part two the distribution changes to an almost equal spread between low (13), moderate (9) and high (11) complexity. A main cause of this shift to higher complexity is low parameterization which results in additional decision points (if-/case-conditions) with every addition of property values (see listing 5 and the following paragraph about parameterization).

Finally, we compare the complexities of the programs of the experimental and the control group. Our expectation was that the students who were taught stepwise refinement would produce programs with functions of lower complexity. Table 7 shows that in the experimental group, more of the implementations are of low (23) than of moderate or high complexity (13). For the control group the relation is the other way around: 24 programs are of moderate or high and only 8 of low complexity. To make sure that this difference is not only caused by the difference in the assignment that the experimental group was explicitly instructed to use functions while the control group was not, we also analyze the complexities of the programs with a depth of the call tree greater than 1 (i.e. the program is split into functions). Table 8 shows that when we look at the students in the control group who chose to use functions for the implementation, there are still more programs of moderate or high complexity (12) than of low complexity (5). For the experimental group the relation stays nearly the same as before (21 programs of low and 10 of moderate or high complexity). The students in the experimental group seem to be more successful in splitting up the problem into pieces of lower complexity.
Listing 5: An example for a function with a clearly defined task but a badly parameterized implementation

```c
// auswerten - Auswerten der eingegebenen Daten
int auswerten(int sportart, int geschlecht, int* fuss_m, int* fuss_w, int*
   hand_m, int* hand_w, int* volley_m, int* volley_w, int* tennis_m, int*
   tennis_w, int* bad_m, int* bad_w, int* gym_m, int* gym_w, int* fuss_k, int*
   hand_k, int* volley_k, int* tennis_k, int* bad_k, int* gym_k) {
    switch(geschlecht) {
    case 0:
        switch(sportart) {
        case 0:
            (*fuss_m)++;
            break;
        case 1:
            (*hand_m)++;
            break;
        case 2:
            (*volley_m)++;
            break;
        case 3:
            (*tennis_m)++;
            break;
        case 4:
            (*bad_m)++;
            break;
        case 5:
            (*gym_m)++;
            break;
        }
        break;
    case 1:
        switch(sportart) {
        case 0:
            (*fuss_w)++;
            break;
        case 1:
            (*hand_w)++;
            break;
        //...
        break;
        }
        break;
    return 1;
    }
```
A $2 \times 2$ contingency table analysis was conducted to verify whether there was an association between group and cyclomatic complexity. For the whole set of programs a significant relationship was present, with chi square $= 10.329$, df $= 1$, $p = 0.001$, in that the experimental group produced more programs of low complexity, while the control group produced more programs of high complexity. The effect size was .390, indicating a medium effect. For the programs with functions (depth of the call tree $\geq 2$) the relationship was also significant, with chi square $= 6.497$, df $= 1$, $p = 0.011$, in that the experimental group produced more programs of low complexity, while the control group produced more programs of high complexity. The effect size was .368, indicating a medium effect.

The numbers show that the students in the experimental group split the problem into smaller pieces. A strong influence is the wording of the assignment. But even with that filtered out, we can see a significant difference between the performance of the two groups.

**Complete**  Most of the modules are complete. There are 25 out of 202 that are not. Almost all of them are incomplete because the implementation of one part of requirements is missing. For most of them, the missing part is the possibility to generate the statistics table for more than one club (11). Also, in part of the programs some output is missing (the prompt for the input or at least part of the table, 9). 2 do not give the user the possibility to stop the input. The most interesting case is a main that only calls the input function eingabe() which looks incomplete, but actually takes over all the functionalities (see listing 4). 20 of the incomplete modules are from 11 students from in the experimental group, 5 from 5 students in the control group. The numbers are too low to draw any conclusions except that the completeness of the modules was not a major problem for the students and that the experimental group did not perform better than the control group.

**Consistent** The only example of inconsistence of a module is the one already mentioned as part of the completeness: The input function that takes over all the functionality of the program and therefore does more than the name predicts (see listing 4). It can be found in the three versions of the program of one student. So overall, at this level of learning to program there is no problem with this quality attribute.

**Utilized** The same applies for the property “utilized”: There is no function that is not utilized. At this level of learning to program there is no problem with this quality attribute.

**Parameterized** The part of main interest to us from the analysis of the properties of the modules is the parameterization. For analyzing the parameterization of the programs, we compare the programs of the students with our expectations of a well-parameterized program as described in section 4.2.2 and summarized in table 9. As mentioned earlier, we want to keep the number of members, sports types and person groups flexible, as these are the changes that we consider as foreseeable\(^6\). For reaching full parameterization of these items, for each of them both suitable data

\(^6\) The list is almost the same as the changes of requirements from part 1 of the assignment to part 2. However, we decided not to include the new requirement to be able to do the statistics for several sports clubs in one run, because we consider that as not foreseeable.
<table>
<thead>
<tr>
<th>change</th>
<th>likely needed datastructures</th>
<th>when to use this datastructure</th>
<th>flow-control constructs to use</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A. max persons</strong></td>
<td>1. integer constant</td>
<td>test during input if we arrived at the max</td>
<td></td>
</tr>
<tr>
<td><strong>B. max sportstypes</strong></td>
<td>1. an array to hold the names</td>
<td>for printing the header of each row</td>
<td>loop over the array while printing rows</td>
</tr>
<tr>
<td></td>
<td>2. an array to hold the counting results</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. for updating the counts after reading one persons data</td>
<td>no separate branches for each sport</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. for printing the counts</td>
<td>loop over the array while printing rows</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. integer constant</td>
<td>a. input checking</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. in loop conditions</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C. max persongroups</strong></td>
<td>1. an array to hold the names</td>
<td>for printing the header of each row</td>
<td>loop over the array while printing columns</td>
</tr>
<tr>
<td></td>
<td>2. an array to hold the counting results</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a. for updating the counts after reading one persons data</td>
<td>no separate branches for each persongroup</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. for printing the counts</td>
<td>loop over the array while printing columns</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. integer constant</td>
<td>a. input checking</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b. in loop conditions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tab. 9: Expected parameterizations in the sports club example
structures and control structures must be used. For example, to reach full parameterization of the sports types it is not enough to use an integer constant for the number of sports types. Also, an array that holds the names of the sports types and an array that holds the counting results are needed. In addition these data structures must be accessed without using hard coded numbers.

For capturing the degree of parameterization, we put these thoughts into a check list consisting of the foreseeable change together with the most obvious data structure to use to be able to react to the change as well as when this data structure should be used an in connection with which control structure (see table 9). In the analysis table, each line of the list is transformed to a criterion and hence a column. The column is filled with either “y” for the expected parameterization, “n” for a different solution and “-” for no implementation at all or a different implementation. We interpret an item as fully parameterized if all the columns belonging to the foreseeable change are filled with “y”. We observed a range of (none-) parameterizations:

A1 The least problematic part was the maximum number of members which could be implemented by fulfilling one criterion: using an integer constant for checking the upper limit when reading the input. Two thirds of the implementations are done that way (46). In other implementations, the upper limit is hard coded (13). And finally there are the ones that do not use an upper limit at all (9) with two different grades of severity: The limit is not needed because the counting is done using variables or the counting result is immediately written in the counting array, so the leak of it is “only” a deviation from the requirements; if the data for all the members is collected in a separate array before counting, the limit is needed because not having it results in writing to a cell of the array which is beyond the initialized range which results in an “array out of bounds”-error.

B More challenging was the parameterization of the sports types. In order to fulfill all the requirements for the parameterization of this item, the implementation needs an array for the names of the sports types, an array for counting and an integer constant for the number of types as well as the according control structures. Only one of the implementations is fully parameterized in the way we expected. Five more implementations from four different students can be added to these if we count in those that did not implement the input check or had a creative solution for printing the headers of the table rows (see below).

B1 One issue is that in most of the programs (57) the names of the sports types and person groups are not stored in an array. Only in five programs from four different students the names are stored in an array and printed using a loop as we expected. An explanation for this might be that the activity was conducted after an introduction to arrays in general, but character arrays had not been introduced yet. It is noteworthy that two students (one from each group) worked around the problem of not knowing how to deal with character arrays and instead printed numbers as labels in the table (see listing 6). From these workarounds we interpret that the students considered the parameterization as important enough to not conform to one of the requirements.

B2 As second part of the parameterization of the sports types, we expected the students to use an array for counting and printing the values. In total, about two thirds are using an array for these purposes (44). However, the use of appropriate control structures is a problem for a big part of them, as described in the next two items.
void fkt_ausgabe(int datenbank[sexes][sports], int male, int female, int children)
{
    printf("Sport \t m\ennlich \t weiblich \t Kinder \t gesamt \n");
    for(i=0; i<sports; i++)
    {
        printf("%d \t %d \t %d \t %d \t %d \n", i, datenbank[0][i],
            datenbank[1][i], datenbank[2][i], datenbank[0][i]+datenbank[1][i]+
            datenbank[2][i]);
    }
    printf(" \n
-------------------------------------------------------------------\n
gesamt \t %d \t %d \t %d \t %d\n", male, female, children,
        male+female+children);
}

Listing 6: An example for working around character array problems and parameterizing sports
types but not person groups by printing a number instead of a name for the sports types

\textbf{B2a} The table shows that in almost two thirds of the programs (42) the parameterization of
the counting is not reached. This group can be split into two groups again: One part is doing the
counting in variables, using one variable for each of the sports type/person group combination (see
listing 5); the other group is using an array for counting, but doing each update of the count in an
extra branch of a switch-case or if-statement (see listing 7). For both of the groups, the counting
of the sports types is not parameterized, as both have to add code for each additional sports type.
Six of the students change from a non-parameterized to a parameterized approach from part one
to part two or part two to part three (half of them switch from variables to an array, half of them
change the way they are using the array).

\textbf{B2b} The other part of the parameterization of the sports type using an array is fulfilled by
even less of the programs. Only 18 implementations loop over the array for printing the results. In
47 implementations, the values are output using a hard-coded variable or the index of the array.\footnote{Note: 3 implementations miss the output.}
We suppose that most of the students who used arrays might have hard coded the indices in the
output, because they hard coded the labels for the sports types anyway. All of the implementations
that use an array containing the sports names or print numbers as row headers also use a loop
for printing. Of those who did not parameterize the sports names, 11 implementations from 7
different students still loop over the array for the output, typically separating the printing of the
row headers and the values (see listing 8), or using the workaround mentioned earlier.
void Statistik(int table[2][MAXMEMBERS], int Stats_t[5][3]){
    int Member;
    for(Member=0; Member<MAXMEMBERS ; Member++){
        switch(table[1][Member]){
            case 1:
                if(table[0][Member]==0)Stats_t[0][0]++;
                else if(table[0][Member]==1)Stats_t[0][1]++;
                break;
            case 2:
                if(table[0][Member]==0)Stats_t[1][0]++;
                else if(table[0][Member]==1)Stats_t[1][1]++;
                break;
            case 3:
                if(table[0][Member]==0)Stats_t[2][0]++;
                else if(table[0][Member]==1)Stats_t[2][1]++;
                break;
            case 4:
                if(table[0][Member]==0)Stats_t[3][0]++;
                else if(table[0][Member]==1)Stats_t[3][1]++;
                break;
        }//end switch
    }//end for
    Stats_t[0][2] = Stats_t[0][0] + Stats_t[0][1];
    Stats_t[1][2] = Stats_t[1][0] + Stats_t[1][1];
    Stats_t[2][2] = Stats_t[2][0] + Stats_t[2][1];
    Stats_t[3][2] = Stats_t[3][0] + Stats_t[3][1];
}//end Statistik

Listing 7: An example for counting in an array but with separate branches
6.1 The Product

```c
/*Ausgabe*/
printf("\t m w gesamt\n");
for(i = 0; i < anzahl; i++){
    switch (i){
        case 0: printf("Fussball "); break;
        case 1: printf("\nHandball "); break;
        case 2: printf("\nVolley "); break;
        case 3: printf("\nTennis "); break;
        /*hier muesste man fuer eine erweiterung weitere verschluesselungen
definieren*/
    }
    for(j = 0; j < 3; j++){
        printf("%d ", werte[i][j]);
    }
}
printf("\ngesamt %d %d %d", werte[anzahl][0], werte[anzahl][1], werte[anzahl][2]);
```

Listing 8: An example for a partly parameterized output, comments by the student

**B3** The third item contributing to the parameterization of the sports types is to keep the number of sports types parameterized. The most obvious way to do that would be by using an integer constant.

**B3a** The integer constant can then be used for checking whether the user’s input is within the valid range of numbers. In most of the programs (53) this feature is not implemented at all. This can be explained by the teachers’ input that at the current stage of learning the students are allowed to expect the user to enter only valid numbers. In 15 programs (from 11 students) the feature is still implemented. 5 of them (from 5 students) use an integer constant for the check, 10 (from 8 students) do not. 5 changes from part one to two: one introduces an input check that is not parameterized, one introduces an input check that is parameterized, three change their approach from non-parameterized to parameterized.

**B3b** As further check for parameterization, we expect the students to use the integer constant as a loop condition in general, rather than hard coding the number of sports. However, it turns out that about half of the implementations do not use loops over the sports types at all (32) which might be explained by the popularity of explicit switch-case- and if-statements in general (see above). Of those who do loop over the sports types (35), more implementations use the integer constant as condition (23) than hard code the number in the condition (13). Six students changed to using a constant in the loop conditions from part one to part two. One of them had not used loops with the sports types as conditions in the first part. The opposite change did not occur.
for (i = 0; i < current; i++) {
    if (Person[i][0] == 0)
        Statistik[Person[i][1]][0]++;
    else if (Person[i][0] == 1)
        Statistik[Person[i][1]][1]++;
    else
        Statistik[Person[i][1]][2]++;
}

Listing 9: An example for partly parameterized counting (sports types parameterized, person groups fixed)

However, there was one student that introduced a loop with the sports types as condition in part two without parameterizing it.

C The requirements for the parameterization of the person groups are analogous to those for the sports types. Accordingly, the patterns of implementations are similar. Overall, we can see a shift to less parameterization, which might be because the number of person groups is lower and with genders given in the beginning it is less foreseeable that more person groups will be added. For example, this can be seen in the counting of the members, with 52 programs not parameterized and 16 parameterized, compared to 42 programs not parameterized and 26 parameterized in this way for the sports types. For both counting the members and printing the results, it is a common pattern to not parameterize the person groups even if the sports types are parameterized. Examples are given in listings 8 and 9.

In general, the students were not very successful in parameterizing the programs. For one part, parameterization is not reached, because they do not have the required knowledge yet. Most striking however is to see how the students are putting into practice what they learned and figuring out how to use data structures, which at that moment still inhibits parameterization.

Loosely coupled Coupling is a problem for only a few students. The functions in the programs of 4 students in the experimental group are coupled while in the control group there are only 2. We do not consider this difference as significant, as in the experimental group there were more students using functions than in the control group, and only if functions are used coupling can happen.

The bad coupling happening the most is “common coupling”. Three candidates in the experimental group passed data between functions in global variables. One student from the control group uses a global variable to control the behavior of one function from another function.

Two other students (one from each group) use another style of programming that might cause problems: They pass a high number of pointers/variables as arguments of the functions. This is a

8 One of the students made a joke during the first part of the assignment about whether it should be possible to enter a person of “neutral” gender because he had an article about a similar topic in the news.
form of stamp coupling, because as the parameters of one function change, the parameters of the other function will most probably have to be changed, too.

We summarize that in general, coupling was not a problematic issue. Problems do or do not appear independent from our intervention.

6.1.3 Sequence

Structured Our expectation is that the experimental group would stick more to the paradigm of structured programming, because they are asked to use structograms for the planning, and structograms enforce structuredness by having blocks with one input and one exit.

The data however proves different. 11 students from the experimental group produce unstructured code, while only 8 from the control group do so. In each of the groups there is one student using the infamous “goto” (listing 10). The rest of the students is exiting loops by either using a “break” statement (listing 15), simulating a break by setting the counter variable to the upper limit, so the loop will be exited (listing 12), or using “return” to exit the function (listing 11). A few of them are using “continue” as well, in combination with “break”. The unstructured solutions are all used for controlling the input. One student is using another break-statement for stopping the counting of the members in addition.

Comparing the results from the experimental group with their structograms, we find that 7 of them had actually planned the stop of the user input in the loop condition, 3 of them state the “break” in the structogram already, and 1 of them does not plan the exit at all.

From that we conclude that the benefit of the concept of structuredness might not have been recognized by the students yet. Even though in many cases their planning already indicates the “good” way of implementing the user input, the students choose the “easier” way in the implementation. However, the reason for that could also be that the students do not know how or hesitate to deal with loops with multiple conditions.

Cohesive There was hardly any variation in the coherence of sequences; all of them were sufficiently cohesive. At this level of learning to program there is no problem with this quality attribute.

6.1.4 Loop

Complete We count loops as “complete” if they are guaranteed to terminate. For that we check whether a termination condition is clearly identifiable and that this condition will be met. Standard for-loops are complete, but also while-loops that show progression towards the termination condition. If user input is expected as termination condition the termination is not guaranteed. However, we do not count this behavior as defect because it is necessary and the termination can be reached by the user.

Most of the loops applied are complete or terminated by user input (only one exception). Overall, at this level of learning to program there is no problem with this quality attribute.
```c
int HoleSportart(int Nummer) {
    int x;
    x = 0;
    printf("# Bitte die Sportart des %d. Sportlers eingeben: #\n", Nummer);
    Eingabe:
    printf("#
    #%c", char(13));
    printf("#
    printf("# ");
    if(scanf_s("%d", &x) != EOF) {
        if(x<0) {
            printf("# Bitte keine negativen Zahlen eingeben! #\n");
            goto Eingabe;
        }
        if(x>3) {
            printf("# Bitte nur Werte zwischen 0 und 3 eingeben! #\n");
            goto Eingabe;  //unstructured
        }
    } else {
        iBreak=1;  //iBreak is a global variable -> coupled
    }
    return x;
}
```

Listing 10: Unstructured with goto and coupled
while(i<MITGLIEDER) {
    printf("hallo, wollen sie ein Mitglied hinzufügen? 1=ja 0= nein\n");
    fflush(stdout);
    scanf("%d",&eingb);
    if(eingb == 1) {
        printf("\nDie Sportart bitte
0=Fussball, 1=Handball, 2=Volleyball, 3=Tennis\n");
        fflush(stdout);
        scanf("%d",&sport);
        eigenschaften[i][0]=sport;
        printf("\nGeschlecht bitte
0=Männlich, 1=Weiblich\n");
        fflush(stdout);
        scanf("%d",&geschl);
        eigenschaften[i][1]=geschl;
        i++;
    } else if(eingb == 0) {
        return;
    } else {
        continue;
    }
}

Listing 11: Unstructured: Exit loop by return

for (i=0; i<anz; i++) {
    x=i+1;
    printf("\nGeben Sie bitte die Sportart fuer das %d. Mitglied ein:\n0 = Fussball\n1 = Handball\n2 = Volleyball\n3 = Tennis\n\nCtrl + Z fuer Abbrechen\n\n", x);
    if (scanf("%d", &mat[i][0]) != EOF) {
        printf("\nGeben Sie bitte das Geschlecht des %d. Mitglieds ein:\n0 = maennlich\n1 = weiblich\n\n", x);
        scanf("%d", &mat[i][1]);
    } else i = anz + 1; //break!
}

Listing 12: Unstructured: Exit loop by simulated break
for(i = 0; i<MAXZEILE; i++) {
    sport[0][i] = -1;
    sport[1][i] = -1;
}

Listing 13: Unresolved

**Initialized and Progressive**  Initialized and Progressive are specializations of the completeness property. We do not observe any (ir)regularities about them. The students handle this issue well. So at this level of learning to program there is no problem with these quality attributes.

**Consistent**  Also, all of the loops are consistent. At this level of learning to program there is no problem with this quality attribute.

**Resolved**  This property is connected to our findings from the parameterization, as it again catches how the students handle the combination of data structures and control structures. For processing arrays which are the only data structure that had been introduced to the students before the assignment, the appropriate control structure to use is a loop. What kind of loop is to be used depends on the situation. If the upper limit is known in advance, a for-loop is most suitable, if the upper limit is not known (e.g. because something is repeated until stopped by the user) a while-loop is most appropriate, if we deal with a combination of both (e.g. filling an array with user input that can be stopped before the end of the array is reached) a for-loop with the right condition is as appropriate as a while-loop.

The other criterion for resolvedness is whether the dimensions of the array are handled appropriately (cf. Dromey’s example for a defect: “use of a single loop to process a two-dimensional array” [Dromey 1995]. Listing [13] shows an example of a loop that is unresolved because of the latter reason. The resolved version of the code is shown in [14]. Similarly, the loop in listing [15] is not resolved, as strictly speaking the scanf statement should be executed in another loop to cover the second dimension of the array. Contrary to that, we track the loop in listing [7] as resolved, as the counting of the members does not require a field by field but a line by line processing of the array.

We did not expect the students to parameterize the number of pieces of information that can be collected. Because of that we expect that the resolvedness according to this aspect will be rather low. This is confirmed by the finding that hardly any input loop is resolved (as in listing [15]). Overall, if loops are used, appropriate choices of the type of loop were made. This is not so surprising, considering that in many cases almost every loop would be counted as appropriate.

We consider the choice whether to use a loop at all as more important which is discussed as part of the parameterization in section [6.1.2]. Another important aspect is whether the loops are used in a structured way which is discussed in the respective part of section [6.1.3].
for(i = 0; i<MAXZEILE; i++) {
    for(j = 0; j<MAXSPALTE; j++) {
        sport[j][i] = -1;
    }
}

Listing 14: Resolved version of listing 13

void WerteEinlesen(int sport[MAXZEILE][MAXSPALTE]){
    int i = 0; //ineffective code
    for(i = 0; i<MAXZEILE; i++) {
        printf("Bitte Werte eingeben nach Schema 'Personengruppe,Sportart' \n 0: Männlich \n 1: Weiblich \n 2: Kind \n 3:Abbruch \n 0: Fussball \n 1: Handball \n 2: Volleyball \n 3: Tennis \n 4: Badminton \n 5: Gymnastik"");
        scanf("%d,%d", &sport[0][i], &sport[1][i]); //unresolved
        if(sport[0][i] == 3) {
            break; //unstructured
        }
    }
}

Listing 15: Unresolved, unstructured and ineffective
Effective There were only a few minor flaws in the effectiveness of the code in loops. There are two students who have defects in this issue. They initialize the counters for the loop in the variable declaration and then again in the for-loop guard (see for example listing [15]). But overall, at this level of learning to program there is no problem with this quality attribute.

6.1.5 Selection

Complete Even though there are many selections, there are hardly any incomplete ones. There is only one exception, in which the children are not counted (and in another function not printed either). Overall, the students were thorough enough in their implementations. At this level of learning to program there is no problem with this quality attribute.

Consistent In the discussion of the general correctness of the program, we mentioned that some of the bugs resulted from selections that did not catch invalid input in the counting and therefore caused incorrect statistics. We do not count these cases as inconsistent, because the students obviously expect “valid” input. For that, all the cases are covered. But in general it would be better style to have a default case that catches the unexpected input.

Other than that, we did not find any inconsistent selections and conclude that this was not a critical issue. At this level of learning to program there is no problem with this quality attribute. Still it would be interesting to discuss about with the students.

Effective A common pattern we observed was the use of a switch-if-structure for checking the value of a variable and then changing the value of the field with the corresponding index in an array rather than using the variable’s value for the index directly. However, we do not mark this as ineffective, because deleting part of the code would change the behavior of the program. This construct is caught as defect of parameterization (section [6.1.2])

Other than that we cannot recognize any significant ineffectivenesses. At this level of learning to program there is no problem with this quality attribute.

6.2 The Process

6.2.1 Documentation

For evaluating the design process, we analyze the documentation by itself and in combination with the code. For the documentation by itself, we use the strategy as described in section [5.2] in order to see how much of what has been taught is used by the students. For each student we check the planning until one requirement is not fulfilled to determine the number of points (see also figure [2]. The more points the planning complies with, the closer the student’s strategy is to what has been taught in class. Because we want to find out how the intervention has influenced the students’ planning process, we focus on the experimental group.

9 Note that this use of the switch-if-structure also makes the code inefficient which is out of the scope of our analysis.
Design  As first step, we filter out the students who did not plan anything at all or at least did not hand in any documentation, resulting in a group of 20 students who did some planning out of 25 students in the experimental group. Next, we check whether the planning was done using stepwise refinement. That can basically be done by planning functions textually or, as we asked in the assignment, in structograms (which we will check in the next step). Most of the students followed the instructions and did at least one refinement step. One student did very detailed planning in one structogram (E110) which we do not see as stepwise refinement.

Two of the 19 designs do not pass the next test, which is whether the stepwise refinement was done using structograms (E2, E3). The designs are both based on only one structogram each. (This structogram is kept general enough to call it a first refinement step which is followed by the refinement that is done when putting this planning into code, so it is not filtered out the previous test.) This leaves 17 designs for the further selection.

A pattern that is immediately recognized in the structograms is that the students have two different strategies of applying stepwise refinement. One group is creating a new diagram for the refinement of each block. The other group is doing the refinement of all blocks in one diagram. Earlier in the course, we had given as feedback on the structogram assignments that each structogram should be complete for its purpose11. One strategy would be to refine the structogram as a whole (e.g. like we did in the first refinement step of the example implementation, refining the information that is needed for all parts of the program), another one would be to take a specific part and refine it in an extra structogram (like we did for the functions). Most of the students follow one of these strategies for the whole design of the program; only one follows different strategies dependent on the refinement steps (E4).

Next, we check whether it is possible to draw the “refinement tree”, i.e. whether each refinement is complete and corresponds to a block on the higher level. This is necessary to fulfill the idea of machines that are like replaceable pearls (see discussion of Dijkstra 1970 and section 2.3.1). This is not the case if a block from the first level is left out (E5) or steps are introduced (E6) in the refinement leading to a design that is not complete on one of the levels and seems to serve more as a correction than as a refinement. In three other designs, blocks are merged and functionalities mixed in the refinement (E7, E8, E9). Problematic is also the refinement of one block (named accordingly) and adding the next block as last step of the refinement (E4, E10). These problems appear with both the students who are doing the refinement in one structogram and those who do it in separate ones. Another incorrectness is keeping the name of the block from the first level in the structogram in the refined structogram in addition to the refinement of the block (E11). However, we do not see that as as severe as the previous mistakes, because it does not endanger the idea of independent machines, and still count it as “refinement tree possible”. In total, this test is passed by 10 designs.

Finally, we assess the simplicity/complexity of the designs. If we take McCabe’s recommendation of considering complexities of >10 as too high, we only find one out of the ten examples that fulfill all the previous requirements and is of high complexity (E12). A closer look reveals that

10 E for example + a number for the student, in order of appearance  
11 That means that if a block is refined, the responsibility of the block should not change.
this example contains the planning of an if- combined with a switch-case statement. If we look at the three designs that are just at the limit of low complexity (E13, E14, E15, complexity 10), we see that they contain similar constructs, and also the ones with a complexity of $\geq 10$ that have been filtered out before. (This means that this ugly design could have been detected in the design phase!) How did the others reach lower complexity? It is important to see that the level of detail of the designs differs. The designs of high complexity define at least which control structure is to be used for the counting and often also how the values will be stored as well. The designs of low complexity do not necessarily do that. Two of them define that the counting will be done in an array but not how (E16 and E17); in one case low complexity is reached on all levels by doing the selections on different refinement levels (E18, using variables); one defines the counting in an array using a loop but in combination with switch-/if-statements which still results in low complexity (E11); only one indicates that the counting will be done in a loop using an array (E19)!

The students seem to have different perceptions of when a reasonable level of planning is reached. Moreover, different ways of planning are apparent in the “language” used in the structograms: Sometimes the steps are very close to natural language (e.g. E14) while sometimes C-specific commands are used (e.g. E15). It is not possible to say what type of student would decide for what kind of design because different explanations are possible. On the one hand, a more experienced student can choose for a very sketchy planning as she would know how to transfer the steps into program code. But for the same student it would be easy to write down the detailed steps. On the other hand, a novice might prefer to stop at a certain point and start coding as the compiler is giving feedback on whether something is working or not. But a novice student might also want to try and prepare the design as neatly as possible before doing the unfamiliar programming work. Concerning this issue, we do not know yet which of these theoretically possible strategies is the most adequate, nor do we have grounds to assign the visible strategies of students to these hypothetical strategies.

Based on these observations, a group of nine students (out of 21 who handed in the documentation) turn out to have realized the planning fully as we had hoped them to. We interpret their implementation of the strategy that they have understood what had been taught in the lecture.

**Design and code** We investigate the connection of design and program code by checking whether the design is recognizable in the code. For that, we look at whether the program structure roughly fits the design. This is true for almost all of the programs. One of the students (E13) deviates from his (still recognizable) design, and describes his choice to do so in the answers to the design questions. We declare only one design (E10) as not recognizable because two steps that were planned separately are merged in the code (input and counting). In addition to that, we observe that some of the designs that are done in one big structogram result in examples of programs with a clear distinction of functions (e.g. E1) while there are students who refine blocks in separate structograms, but implement these blocks as part of one function (e.g. E17). We summarize that overall, the students did really write their programs based on the planning. The splitting up into functions happens independently from the splitting up into structograms, while possibly still conforming to the refinement tree.
6.2 The Process

<table>
<thead>
<tr>
<th>Group</th>
<th>Low (1-10)</th>
<th>Moderate/High (&gt;10)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEXP</td>
<td>8</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>NCON</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Total</td>
<td>12</td>
<td>5</td>
<td>17</td>
</tr>
</tbody>
</table>

Tab. 10: Overview of the maximum complexities for the new experimental group (NEXP) and the new control group (NCON)

Of the students who realized the planning ideally (9 students) we form another experimental group. By comparing this group to the rest of the experimental group which serves as new control group (11 students) we want to see if there is a difference in the software quality and changeability between the group who fully understood and implemented the taught strategy and those who have not. We hardly see any difference between the groups. Concerning the depth of the call tree, we note that the new control group consistently uses functions with a call tree of depth 2. The new experimental group shows more variation in that: 3 students use one big main, 4 make a call tree of depth 2, and 2 make a call tree of depth 3. One of the students using a call tree of depth 3 does it a balanced way, whereas the other is overloading the input function with calling also the rest of the functions; we have both the successful and the unsuccessful decomposition present in that group. Table 10 shows that most of the students in the new experimental group have programs of low complexity (8 out of 9) in part one. The control group half of the programs for part one are of low complexity (4), while the other half is of high complexity (4). While the numbers are too low for being of statistical significance, they confirm the results from the general comparison of complexities: The taught strategy helped the students to decompose the problem into parts of lower complexity, even more for those who fully implemented it. For the parameterization, we do not see a convincing difference between the groups, just like in the general comparison.

We conclude that the nine students that we had selected learned the intended design strategy but it is not visible (yet) that this strategy helps them to take better decisions. Their learning of control structures and how to use them has a bigger influence on the quality of the resulting code. Following up on this issue, we could pose a new research question: Can stepwise refinement be used to make the choice of control structures and data structures more consciously? We suppose that this is a point that will only work if the teacher is aware of it and includes it in the teaching.

Comparing these results to the findings from the analysis of the software quality of the end products by itself, we infer that the experimental group performed better with respect of building programs of lower complexity because for them the planning was stressed more. Which strategy they followed in detail did not make a major difference.

6.2.2 Changes

We measure the changes between part 1 and 2 of the assignment by counting the locations where changes were made for one requirement and the number of lines changed in general. One location is a block of code or a structure. Changes in a switch/case-structure would count for one location,
for example. In doubt we distinguish conceptually different changes, e.g. changes of variables or adding case statements.

We do not find any pattern in the number of locations that were be changed for each requirement. For comparing the number of lines changed, we only look at those students who at least changed the sports types, person groups and number of clubs, giving us a group of 18 students\textsuperscript{12}. The mean number of changed lines is 69.5 with a standard deviation of 29.53.

We want to find out what coding style the students with the top changeability used. For that, we investigate the 9 who changed less lines than the average. 5 of these students are from the experimental group, 4 from the control group. 7 use an array for the counting results for both the sports types and the person groups, one is using separate arrays for counting the person groups (rank 8) and one is using variables (rank 6). 6 do not use separate branches for the sports types, 4 do not branch for the person groups. 5 of the top group are implementing everything in one main, 3 are using functions with a call tree of 2 and one with a call tree of 1. In comparison to that, in the other group (8 students) only 2 are using arrays for counting and both of them do that in connection with separate branches. Like the top group, the students in this group choose to implement the program without functions (2) or with functions (4 with depth of call tree 2, 2 with depth of call tree 3). The mean maximum cyclomatic complexity for the two groups differ: The top-ranked ones have a mean of 8.44, the others a mean of 13.63. Accordingly, the change of complexity differs with the top group having a mean increase of 2.78 and the others having a mean of 11.75.

We conclude that the main factor for reaching good changeability, if measured by the number of lines changed, was parameterization, as the majority of the top group parameterized their programs better than the others. Also, the programs of low maximum complexity were more changeable and, in line with the number of lines changed, the complexity increase is lower for the top group than for the rest of the students. Submissive was the decision whether to use functions or not. This again shows that the parameterization which is dependent on the selection of an appropriate data structure with the according control structure, is the major issue that the students are dealing with.

Looking at the changeability in connection with the design process, we observe that 4 of the 5 students from the experimental group who have a low number of changes are in our top strategy implementation group. The one student from the top strategy implementation group who turns out to have many changes already used a switch-case-construct in his design, so the complexity was just at the limit. This is another indication that the designs could be used for discussing the choice of the data structure and control structure with the students.

### 6.2.3 The Students' Reflections

We analyze the students’ reflections on whether there are points when a student recognizes a bad design. Due to time limits, not all of the students handed in answers to that part of the questions. But we can see some light bulb moments from those who did.

\textsuperscript{12} We do not consider the change of the number of clubs as an essential part of the assignment. But it is a major influence on the number of lines changed.
Some rather general comments:

- “was not good, only max number optimal” (E8, experimental group)
- “Changes could be done easily, but the code could be made more changeable.” (E4, experimental group)
- “No [I did not plan the changes], because I did not have the idea to solve the task using functions. With functions, some things would have been easier, I think. [Changeability] needs to be revised.” (E20, control group)
- “pay more attention to dynamic changes next time” (E21, control group)

About the parameterization in general:

- “very low changeability → programmed many things too statically (statistics, output), so the changes were very difficult for adding the child” (E22, experimental group)
- “many changes for child because gender not defined using ‘define’” (E23, control group)
- “It was quite easy to change the code, only a few changes were needed. Because of the outsourcing of Anz_Sportarten, Anz_Gruppen future changes will be even easier” (E24, control group, sets the numbers of sports types and person groups in integer constants in part 2)

Students improving the parameterization:

- “Overall not very changeable because of the late use of a multi-dimensional array. Basic structure of function ‘process’ wrong for that” (E13, experimental group, referring to a big switch-case construct he used in the early versions, switches from variables to array for counting).
- “original code very inefficient, but by the revision significantly more changeable” (E18, experimental group, uses array for counting and gets rid of a switch-case statement in the second version).
- “now the code is easy to change, but at first, changes were quite difficult” (E25, control group, switches from no parameterization at all to complete parameterization)

Some students do recognize flaws, but do not see all of them:

- “Conclusion: Planned changes easy to realize; use more functions (for storing in array!); generating the table automatically would have saved LOTS of work.” (E26, control group) He is doing the counting in both of the versions of the program in an array, using a switch-case construct.
- “[Number of s]ports types/person groups should have been set in variables → output in loop (variable for sports types/persons), code could have been more changeable (more loops, variables).” (E27, control group) He also does the counting in an array, but using a switch-case construct.
• One student made the conscious decision to not make the code configurable in the first place. In the end, he writes:

  – “My original code was not changeable at all, as I said. In opposite to that, the current code is quite changeable. The next step would be to save the sports types and person groups in a club specific file. Doing that, you could also save the statistics there, so that only changes would have to be entered.” (E15, experimental group)

• And one thinks there is technically no way of improving the code:

  – “I think it was changeable in any case because of the array, but you cannot avoid the effort of writing the output.” (E28, control group) He also does not recognize that his switch-case construct for counting is not optimal.

• Mixed views:

  – “Programmed relatively generally, so that changes were extensive, but fast to do. The only problem was the high number of variables to keep the overview → original code well changeable and clear” (E10, experimental group)

  – “orientation easy, changeability low because it needs to be changed in many locations” (E29, control group)

• And at last an observation about an issue that we do not analyze explicitly:

  – “I would not use 2 arrays again, but increase the number in the right field of the statistics table directly after the input. I first expected that the table with the players should be kept. That is why my program consists of 2 arrays which of course make the program very large (and probably slow). The problem then was that I had many sub-programs which I had to pass the arrays to, which led to many errors and loss of time.” (E5, experimental group)

There was also a number of students who commented on the changeability, but only in a positive way. Not all of them had developed a changeable solution, in our opinion.

For a group of students we see learning happening. They recognize that their first designs were improvable. Using the “right” control structure with the array again seems to be a problem, as some of the students who are critical about their code in other respects do not mention this issue. The answer of the student who thinks that the output cannot be implemented in a better way than hard coding the table confirms that the students’ solutions are limited by their state of knowledge.
7 Conclusions and Discussion

7.1 Conclusions

RQ1: How does the teaching of “stepwise refinement” as a design strategy enable students to decompose a problem and react to changing constraints?

The teaching of stepwise refinement helped the students to decompose the problem and produce programs with modules of lower complexity, which is shown by the comparison of the cyclomatic complexities of the programs of the experimental and the control group. The students who fully implemented the taught strategy, and therefore were the most consistent in applying stepwise refinement, were better at producing programs of low complexity than the rest of the experimental group. This shows that it is not only the focus on planning, but also the understanding and application of the stepwise refinement as a strategy, that mattered for separating out parts of the problem into pieces of lower complexity.

Looking at the students who had the least changes in their programs after the requirements for the program were broadened, we see that the mean of the complexities of the programs of the top group is lower than the mean of the rest. This confirms the expectation that the attribute complexity influences the changeability of the program.

The other attribute which we analyzed that relates to changeability is the parameterization of the programs. The importance of parameterization for the changeability is confirmed by the observation that the parameterization of the programs of the top changeability group is rather good, while in the entire groups of students the parameterization is generally poor. We cannot recognize any influence of the strategy taught on parameterization. For this attribute, the students’ learning of the basic ideas of programming, such as the use of data structures and the interaction with the use of control structures, dominates any possible effect of the intervention.

In summary, the teaching of stepwise refinement as a design strategy enables students to decompose a problem into pieces of lower complexity. The lower complexity also contributes to better changeability and therewith helps the students to react to changing constraints.

RQ2: How does the use of documentation and visualization (writing down stepwise refinement, using structograms) support the process of learning to design software?

Firstly, we consider the influence of documentation and visualization on the application of the stepwise refinement: On the one hand, the structograms of many students who did the stepwise refinement in structograms cannot be ordered in a refinement tree. This is an indication that the structograms did not necessarily help the students with applying the stepwise refinement consistently. On the other hand, the majority of the students who produced structograms with a reasonable refinement tree have as a result both structograms and programs of low complexity for the first part of the assignment. Therefore, although the structograms did not clearly support (nor hinder) the stepwise refinement, both parts of the taught strategy together had a clear influence in helping the students to reach programs of lower complexity. We interpret the role of the structograms as being tools for the students to capture and evaluate their design decisions.
Further, we see that most of the students roughly follow their planning. For all students who made structograms except one, the contents of their structograms are clearly recognizable in their code. The one student for whom we observe big changes in the program structure took this decision consciously. Although the contents of the structograms clearly are recognizable in the code, the splitting up of the program into functions appears as largely independent of the structograms. Seeing the quality of the result (complexity, coherence), we believe that students who did not map one block in the structogram to one function took this decision consciously, too. From this we conclude that documentation and visualization support students in making conscious decisions in the design process.

The use of documentation and visualization (writing down stepwise refinement, using structograms) supports the process of learning to design software by serving as a tool for capturing and evaluating the design decisions students are making and by encouraging students to make their decisions consciously.

**RQ3: Do stepwise refinement and visualization influence the quality of the resulting code?**

In answering question 3 we looked at all quality criteria in our model (section [5.1.2]) that had not been covered by question 1. For the attributes complete, consistent, utilized, loosely coupled, cohesive, initialized, progressive, resolved and effective we did not find any influence of the intervention nor did we expect it. For the attributes correctness and structuredness, a positive influence was expected but not found.

Concerning correctness, the analysis of the programs as the final product is not very promising. Both groups produced a rather low number of programs that do not contain bugs. We take this as an indication that the students were busy putting the pieces together and so did not get to the point of paying attention to the details. This could also be a timing issue, as the time for the assignments in the lab was rather short. Moreover, the functionality of the programs is often limited by compromises that the students were granted due to their learning stage (e.g. expecting valid inputs) or which they applied on their own account (e.g. setting the maximum number of members low in order to stop the input without having to program user interaction).

In both of the groups there were students who violated structuredness, almost exclusively in the input part. In the structograms of the respective students in the experimental group, we find that many of them actually planned the loops in a structured way but did not implement that detail of the design in the code. We conclude that either the benefits of structured programming are not clear to the students, and they give up the structuredness for an easier implementation, or the students have problems with implementing the original design (e.g. programming loops with multiple conditions).

Summarizing, stepwise refinement and visualization do influence the quality of the resulting code concerning the depth of the call tree and the complexity as discussed for research question 1. For the attributes parameterization, correctness and structuredness, however, the learning of the programming language itself is dominant.
7.2 Discussion

Decision on the hypothesis

Based on the answers to the research questions we conclude that fostering the building of a suitable meta-schema for program design supports students in learning to develop flexible programs even in a first semester programming course. However, we do not have evidence yet that fostering the building of a suitable meta-schema for program design supports students in learning to develop correct programs.

7.2 Discussion

7.2.1 Evaluation of the research setup

Since we conducted the study as “action research” in an on-going course, a major decision we made during the study was to take only one snapshot in the learning process of the students. In the sports club assignment the students of both groups had the opportunity to practice planning using either the taught strategy (experimental group) or an intuitive strategy (control group) for the first time. Because we used this activity as the main test of the effectiveness of the intervention, the learning phase and testing were mixed here. The disadvantage of this is that the purposes are contradictory. On the one hand, from the teacher’s perspective, we wanted to offer the students scaffolding in the learning of the strategy; on the other, from the researcher’s perspective, we wanted to influence them as little as possible in order to see how they implement the approach on their own account. This tension between learning and testing could have been evaded by selecting one of the later programming assignments as our test situation. However, we did not choose that option because later lab assignments did not offer the students as much freedom for design decisions and therefore were not as meaningful for our research.

Another point that should be mentioned is the involvement of the researcher in the course. In the first part of the activity, the researcher taught about structograms and stepwise refinement in the experimental group, while the teaching of the corresponding part was done by the cooperating teacher for the control group. Moreover, the researcher was involved in the course as a lab assistant, for the experimental group throughout the course (because of the group size) and for the control group for the lab sessions starting with the main intervention. The drawback of this involvement is a possible distortion of the results; the benefit is that the researcher could observe the students more closely.

7.2.2 The findings

Possible extension of this research  It is important to note that the students were still a long way from using a “(meta-)schema”. If they were using one they would have had a solution pattern available which they could have instantiated for the current situation, i.e. the process of designing a piece of software. As described earlier, a problem in the research setup is that the students were practicing the use of the strategy and visualization for the first time in connection with programming. Because of this, we can only observe a part of them fully applying what had been taught. An issue for further research would be to see how students’ performance changes
as they gain more practice in the routine of using the strategy and visualization and progress in constructing a meta-schema.

Another dimension for extending the observation would be to look at more software quality attributes and more closely at some of the ones we already analyzed. One group of attributes could be the descriptive properties, including the documentation of the code which might be influenced by the planning and is important for the understandability of the code. Also, we predict that other attributes, e.g. cohesion, will play a more critical role as the students progress and the assignments get larger.

Further, we recommend to find out about students’ perceptions of the intervention. An important question would be whether the explicit teaching of planning strategies decreases the load that planning puts on students’ shoulders or contributes to the “shock” mentioned by du Boulay (Robins et al., 2003 also see section 2.1). In addition, it would be interesting to see how the students’ previous programming experience influences the effects of the intervention. We think that these points deserve special attention because they will certainly influence the students’ perseverance and motivation in their studies, and in the long term they will influence retention. Student retention is a problematic issue in computer science education (see for example the contributions to Rodger et al., 2008). New insights in this direction would certainly be of interest to a broad audience, including of course the students themselves. Promising results regarding student retention are reported by Moskal et al. (2004) as a consequence of a course that also contained the fostering of meta-schemas in the broadest sense (see the discussion about curriculum issues below).

Curriculum issues Our research project makes a case for a curriculum that is different from the “conventional” curriculum of an introduction to programming (Robins et al., 2003). The most common CS1 curriculum is mainly based on the features of the programming language. Our intervention integrated the development of a meta-schema by encouraging design activities in such a curriculum and is strongly related to the “movement in computing and informatics education [that] considers programming to be an application of skills in problem-solving” (Pears et al., 2007).

Similar research was done as reported in Cooper, Dann, & Pausch (2003) and Moskal, Lurie, & Cooper (2004). There, an implementation of a course which was offered in addition to the CS1 course and which was designed for students with weak programming experience is described. The concept of this course, like ours, stressed the role of language-independent problem-solving skills. The evaluation of the proof-of-concept study (Moskal, Lurie, & Cooper, 2004) shows that the course improved the students’ performance in CS1 as well as their retention within and attitude toward computer science. However, this research differs from ours in that the students worked in a programming environment that was only used in the pre-CS1 course before the “real” programming language was learned. Moreover, an important element of the study was the introduction of object-oriented concepts. In contrast, our students transferred their planning directly to the programming language they were learning, and we worked with a procedural programming language.

Another related study was conducted by Lane & VanLehn (2005) who used a natural language tutor to improve the plan decomposition and composition behavior of students. The tutor presents

\[\text{Cooper et al. (2003)}\] propose a new programming course which is evaluated in Moskal et al. (2004).
a pre-defined decomposition of a problem from the general goal to a schema of the solution and finally the plan as pseudo code. After the tutoring session the students went and implemented the solution in program code. The students in the experimental group were interacting with the tutor in a discussion in which the tutor elicited the given solution from the students. These students were compared with a group of students who only read about the same problem decomposition. An interesting part of this research is that the researchers look at the process of coding the final implementation. One of the results is that the students who interacted with the tutoring system “worked more at the level of schemas and plans rather than by the line-by-line perspective typically taken by novices”. Another result is that in the experimental group, “students preferred the more abstract, conversational style representation of programming knowledge over the more concrete, pseudocode representation.” However, students did not perform better in the decomposition on a written test that was conducted after the assignments prepared by the tutoring sessions. This leads the authors to the conclusion that the students should be more involved in the planning rather than being presented a given solution, which confirms our proposition that students should be engaged in planning activities in order to practice problem solving.

For further research on meta-schemas, we think it would be interesting to try out the effects of teaching the meta-schema at different times, e.g. only in the beginning of or as preparation for the CS1 course, or including it throughout the activities in order to encourage the students to build a routine in the development process. Finally there is the question whether, and if so how, students will continue to use the taught planning strategy even when they are not instructed to do so.

There are also supporters of the stance that novice programmers should not be confronted with design aspects. A statement is for example to be found in “Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on Levels of Cognitive Development” [Buck & Stucki 2000]. The authors base their argumentation on Bloom’s (original) taxonomy, stressing that the teaching should be based on the cognitive development of the students. It is noteworthy that two years later a revision of Bloom’s taxonomy was published that puts “metacognitive knowledge” into an independent dimension and by that recognizes that in parallel with learning details learning of metacognitive strategies can occur [Krathwohl 2002].

Beyond the meta-schema aspect, our research results confirm that “basic program planning rather than specific language features is the main source of difficulty” for the students [Robins et al. 2003]. What is the difference between using a loop with multiple conditions or using a break? What is the advantage of using an array over using variables? And how should an array be dealt with? These are examples for questions that can be found in the code and should be addressed before the students are expected to develop their own programs in “real life” but are not answered by teaching the students the notation of the programming language. Neither are they answered by the bare planning strategy alone. We agree that we should address “how to put the pieces together” (Spohrer and Soloway, cited in Robins et al. 2003), and we suggest that discussing the students’ planning decisions and giving them feedback on their work should be

---

14 The teaching can happen in different ways, including lecturing, but also assignments, discussions etc.
integrated in the course. This basically means that we also support the development of specialized
design schemas (see section 2.2).

Taking all our findings together, our ideal first programming course is aware of and fosters
both a meta-schema for design and promotes the development of specialized design schemas. How
this can be realized is an open question, but the discussion should certainly include the division
of tasks between the first course and more advanced courses like “algorithms and data structures”
and “software engineering”. Usually both of these subjects are treated later, separately from the
introductory programming course.

Cognitive tools  We consider the role of the structograms a particularly interesting topic for
further research, as their usefulness seems to be a recurring point of discussion between teachers
who are convinced that structograms are important and students who think they are not. Because
of this we need to provide sound reasons for and against applying the technique.

The structograms serve as what Jonassen (2003) calls a “cognitive tool”. Cognitive tools are
tools for externalizing the student’s idea of a problem and its solution (internal representation).
One advantage of using cognitive tools, according to Jonassen, is that they can be used to off-load
memory tasks. This goes along with our expectations that the use of structograms as part of
building a meta-schema would help students to write down and keep track of the design process
(see section 2.3.2), which as a consequence would lead to fewer breakdowns due to cognitive
limitations (Guindon et al. 1987) discussed in section 2.1. In our study many students did
not get to the point of paying attention to the details, which does not confirm this expectation.
However, timing constraints might have been a major influence in this respect. For a follow-up
study we would recommend to give the students more time, and in addition see whether the off-
loading of memory tasks will pay back more in the long run after students have established a
routine of using structograms.

Jonassen (2003) also provides insight on how the structograms could be beneficial by arguing
that students need to learn to represent problems in more than one way in order to be able to
transfer their problem solving skills. In science this usually means that a qualitative representation
must be developed before the problem is solved quantitatively. For programming students we
interpret this to mean that if we let students develop a representation of the problem, it will help
them to transfer their experience to other problems and even to other programming languages.
This is in line with our reasoning about the development of a general meta-schema for design, but
was not part of our research. A question for follow up work would be how the use of structograms
helps students to transfer their knowledge to other domains and programming languages.

Moreover, a possible field of investigation is to compare structograms to other forms of vi-
sualizations (e.g. UML diagrams) for their suitability, as structograms have been criticized as
outdated. We do not think that this is a valid argument, however, because UML does not offer
the possibility to define the internal structure of methods. Further, the use of different software
tools would be a topic for further research, e.g. the use of software that supports the stepwise
refinement in structograms, or software that connects the diagrams to program code.
Research using cognitive tools for teaching programming has been done in a study described by Cooper et al. (2003) and Moskal et al. (2004) who used a 3D graphics environment as simplified environment for an introduction to object-oriented concepts and the development of problem-solving skills. Roy (2006) presents an environment that connects pseudocode to “real” program code and by this encourages the students to develop and understand code in close connection to the planning. Some of our results suggest that a closer connection between planning and code could be beneficial.

In addition to supporting the learning process, the designs produced by the students can be used by the teacher. One use can be for classroom discussion about the quality of designs; another can be in connection with the program code to get insight into problems students have in translating the design into code.

**Correctness and language** Although Dijkstra’s Notes on Structured Programming (Dijkstra, 1970) were a key source of inspiration for our research project, we made the choice not to put much emphasis on reasoning about correctness in the course. We decided this because an introduction to correctness proofs would have been beyond the scope of the course as an introduction to the language C. Instead, we explained how drawing structograms for each of the needed refinement levels makes it easier to check (using common sense) the correctness of the design at each level because of the better overview and lower complexity.

As we observed in connection with the complexity of the structograms, the students developed structograms using different styles of language. Some describe the design using very free, natural language in the structograms of all levels of refinement; others are very precise and use pseudocode which is very close to the “real” solution. We suspect that the level of preciseness in the structograms might have influence on the correctness of the code. We propose to analyze the language styles used in planning, together with their consequences and the students’ reasons for choosing them for further research.

**7.2.3 Further research**

Our research project has shown that fostering the development of a meta-schema, by teaching a design strategy and visualization, helped novice programming students in learning to develop flexible programs. For further research we propose to

- observe how the performance of students constructing a meta-schema changes over time;
- broaden the analysis of software quality attributes;
- investigate the students’ perceptions of the intervention;
- analyze how the students’ previous programming experience influences the effects of the intervention;
- try out the effects of teaching the meta-schema at different times;
- inquire whether, and if so, how students will continue to use the taught planning strategy when they are not instructed to do so;
– investigate whether the off-loading of memory tasks pays back more in the long run when the students have established a routine of using structograms;

– find out how the use of structograms helps students to transfer their knowledge to other domains and programming languages;

– compare structograms and other forms of visualizations for their suitability;

– explore the use of different software tools, e.g. the use of software that supports the stepwise refinement in structograms, or software that connects the diagrams to program code;

– study the language styles used in planning, together with their consequences, and the students’ reasons for choosing them for further research.

This study has shown that the learning of a first programming language can be usefully combined with learning other parts of the software development process, particularly planning. A revision of the order and emphasis of topics already given in the computer science curriculum clearly has potential, and further research could enlarge the already shown benefits.
Appendices
A Teaching

As described in section 4.2 the teaching part of the intervention consisted of one session about stepwise refinement and structograms for the experimental group and one session about software quality for both groups. Here, we include the teaching materials used for the intervention.

A.1 Structograms and Stepwise Refinement (Experimental Group)

For the first part of the teaching, which was only done for the experimental group, we include the teaching notes. The contents were presented using powerpoint-slides. Except of the “Koche Pulverkaffee” example the notes were included in the syllabus.
Darstellung von Algorithmen

Der Mensch hat gegenüber der Maschine die vorteilhafte Eigenschaft, ein "fehlertolerantes" System zu sein, das auch mit weniger präzisen oder gar fehlerhaften Formulierungen zu einer brauchbaren Problemlösung kommt. Algorithmen für einen Rechner müssen exakt formuliert sein und obige Eigenschaften besitzen. Die Formulierung eines Algorithmus soll möglichst unabhängig von einem bestimmten Rechner, Betriebssystem oder einer Programmiersprache sein.

Die Formulierung eines Algorithmus soll möglichst unabhängig von einem bestimmten Rechner, Betriebssystem oder einer Programmiersprache sein.

Zur Darstellung von Algorithmen sind in der Informatik folgende Diagramme üblich:
- Programmablaufplan (PAP, Flussdiagramm)
- Struktogramm (Nassi-Shneiderman-Diagramm seit 1973)
- UML (unified modeling language) für objektorientierte Sprachen
- Pseudocode

Struktogramme und die schrittweise Verfeinerung


Dijkstra beschreibt in seinen "Notes on Structured Programming" die Methode der "schrittweisen Verfeinerung" (stepwise refinement). Er sieht ein Programm als eine Perlenkette. Jede Perle ist eine "Maschine", die eine bestimmte Aufgabe hat. Wie diese Maschine funktioniert, ist zunächst nicht interessant, sondern wird Schritt für Schritt verfeinert, bis eine Konkretisierung erreicht ist, die die ausführende Maschine (oder Person) versteht.

Das Ziel der schrittweisen Verfeinerung ist es, dem Programmierer zu helfen, den Überblick über das Programm zu behalten. Auf jeder Ebene soll die Verfeinerung nur so weit gehen, dass man immer noch einfach nachvollziehen kann, dass das Programm korrekt ist, es also seine Aufgabe fehlerfrei erfüllt.

Ein einfaches, nicht-technisches Beispiel:

Der Algorithmus "Koche Pulverkaffee"
1. Koche Wasser
2. Gib Kaffeepulver in die Tasse
3. Fülle Wasser in die Tasse

Je nachdem, für wen die Anweisung geschrieben ist, ist sie auf diesem Level vermutlich schon ausreichend. Manchmal ist eine genauere Beschreibung aber hilfreich.
Die erste Verfeinerung

{Zuerst Wasser kochen}
(1.1) Fülle Wasserkessel
(1.2) Schalte Herdplatte an
(1.3) Warte, bis das Wasser kocht
(1.4) Schalte Herdplatte aus

{Gib Kaffeepulver in die Tasse}
(2.1) öffne Kaffeeglas
(2.2) Entnehme einen Lößfel Kaffee
(2.3) Kippe Lößfel in die Tasse
(2.4) Schließe Kaffeeglas

{Fülle Wasser in die Tasse}
(3.1) Gieße Wasser aus dem Kessel in die Tasse, bis die Tasse voll ist

Natürlich kann man den Prozess auch noch genauer beschreiben.

Die zweite Verfeinerung

{Zuerst Wasser kochen}
(1.1.1.) Stelle Kessel unter Wasserhahn
(1.1.2.) Drehe Wasserhahn auf
(1.1.3.) Warte, bis Kessel voll ist
(1.1.4.) Drehe Wasserhahn zu
(1.2.) Schalte Herdplatte an
(1.3.) Warte, bis das Wasser kocht
(1.4.) Schalte Herdplatte aus

{Gib Kaffeepulver in die Tasse}
(2.1.1.) Nehme Kaffeeglas aus dem Fach
(2.1.2.) Entferne Deckel vom Kaffeeglas
(2.2) Entnehme einen Lößfel Kaffee
(2.3.) Kippe Lößfel in die Tasse
(2.4.1.) Schraube Deckel auf das Kaffeeglas
(2.4.2.) Stelle Kaffeeglas in das Fach zurück

{Fülle Wasser in die Tasse}
(3.1.) Gieße Wasser aus dem Kessel in die Tasse, bis die Tasse voll ist

Da wir nun annehmen, dass jeder, der die Beschreibung liest, sie verstehen würde und sich einen Pulverkaffee kochen könnte, ist sie fertig.
A.1 Structograms and Stepwise Refinement

Grafisch kann man die Verfeinerungen als Baum darstellen:

Kontrollelemente
dienen dazu, den Ablauf eines Algorithmus zu steuern. Dies sind:

Sequenz (Folge)

<table>
<thead>
<tr>
<th>Anweisung 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anweisung 2</td>
</tr>
<tr>
<td>Anweisung n</td>
</tr>
</tbody>
</table>

Selektion (Auswahl)

Einfache Auswahl

<table>
<thead>
<tr>
<th>ja</th>
<th>nein</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bedingung</td>
<td></td>
</tr>
<tr>
<td>Anweisungsblock 1</td>
<td></td>
</tr>
</tbody>
</table>

Mehrfach-/Fallauswahl

<table>
<thead>
<tr>
<th>Wert(ebereich) 1</th>
<th>Wert(ebereich) 2</th>
<th>Wert(ebereich) 3</th>
<th>Wert(ebereich) n</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anweisungsblock 1</td>
<td>Anweisungsblock 2</td>
<td>Anweisungsblock 3</td>
<td>Anweisungsblock n</td>
<td>sonst</td>
</tr>
</tbody>
</table>
Iteration (Wiederholung)
Kopfgesteuerte Schleife

<table>
<thead>
<tr>
<th>solange Bedingung wahr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anweisungs- block 1</td>
</tr>
</tbody>
</table>

Fußgesteuerte Schleife

<table>
<thead>
<tr>
<th>Anweisungs- block 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>bis Bedingung wahr</td>
</tr>
</tbody>
</table>

Aufruf anderer Algorithmen

Programm-, Prozedur-, oder Funktionsname (evtl. mit Wertübergabe)

Ein Beispiel

Vergleichen von Intervallen

Es soll geprüft werden, ob zwei Intervalle \( A (a, b) \) und \( B (c, d) \) gleich sind, einander enthalten oder sich überschneiden. Es können beliebig viele Paare eingegeben werden, bis das Programm beendet wird.

Der Algorithmus “Intervallvergleich”

Wiederhole
1. Eingabe der Intervalle
2. Test der Intervalle und Ausgabe
3. bis das Programm beendet wird.

Erste Verfeinerung

1.1 Eingabe Intervall A
1.2 Eingabe Intervall B

2.1 Sind A und B gleich?
2.2 Enthält A B?
2.3 Enthält B A?
2.4 überschneiden A und B sich?

In der zweiten Verfeinerung wird entschieden wie die Details der benötigten Prüfungen aussehen. Wenn man nur die Verfeinerung des zweiten Schrittes betrachtet, sieht der Baum aus wie folgt:
A.2 Software Quality

For both the experimental and the control group, the lesson about software quality started with an introduction to software quality in general presented by the researcher (see section A.2.1). The lecture included the development of a collection of expectations of quality from different stakeholders on the blackboard in interaction with students (table 11) followed by a discussion of how a programmer can contribute to software quality (figure 3). For the experimental group, an extra part about stepwise refinement and structograms concluded the lesson (see section A.2.2). We reproduce the powerpoint slides used during these lectures.

A.2.1 Both Groups

Definitionen (1)

Software (Brockhaus)
Sammel-Bez. für Programme, die für den Betrieb von Rechensystemen (z.B. Computern) zur Verfügung stehen, einschließlich der dazugehörigen Dokumentation.
Definitionen (2)

Qualität (nach DIN 55350)

ist die Gesamtheit von Eigenschaften und Merkmalen eines Produkts oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung bestimmter Erfordernisse beziehen.

Perspektiven

- Auftraggeber
- Programmierer
- Anwender
**SWQ-Merkmale**

(ISO/IEC 9126, DIN 66272)

- Funktionalität
- Zuverlässigkeit
- Benutzbarkeit
- Effizienz
- Änderbarkeit
- Übertragbarkeit


---

**SWQ im Entwicklungsprozess**

Software-Entwicklung: In verschiedenen Phasen wird schrittweise die SW entwickelt (idealer Fall).

Software-Management: Ein entsprechendes SW-Projekt kontrolliert die einzelnen Schritte der SW-Entwicklung.

Software-Qualitätsmanagement: Entsprechende Prozesse garantieren die Qualität der gesamten SW-Entwicklung.

Quelle: Wache, 2007
<table>
<thead>
<tr>
<th>Auftraggeber</th>
<th>Programmierer</th>
<th>Anwender</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effizienz, Sparen von</td>
<td>Änderbarkeit, Flexibilität,</td>
<td>Bedienbarkeit</td>
</tr>
<tr>
<td>Ressourcen</td>
<td>Erweiterbarkeit</td>
<td></td>
</tr>
<tr>
<td>Funktionalität, Aufgabe</td>
<td>Wartbarkeit, Verständlichkeit,</td>
<td>Leicht zu lernen</td>
</tr>
<tr>
<td>erfüllen</td>
<td>übersichtlicher Code</td>
<td></td>
</tr>
<tr>
<td>Portabilität</td>
<td>Effizienz</td>
<td>Benutzerfreundlichkeit, Ergonomie</td>
</tr>
<tr>
<td>Benutzerfreundlichkeit</td>
<td>Dokumentation (technisch)</td>
<td>Verlässlichkeit, Robustheit,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stabilität</td>
</tr>
<tr>
<td>Sicherheit</td>
<td></td>
<td>Schnelligkeit, Antwortzeit</td>
</tr>
<tr>
<td>Gewinn, Zeitsparnis</td>
<td></td>
<td>Erfüllung aller Aufgaben,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Funktionalität</td>
</tr>
<tr>
<td>Verlässlichkeit, Robustheit</td>
<td></td>
<td>Kundenservice</td>
</tr>
<tr>
<td>(Erweiterbarkeit)</td>
<td></td>
<td>Standards einhalten</td>
</tr>
<tr>
<td>Kompatibilität mit</td>
<td></td>
<td>Gute Dokumentation</td>
</tr>
<tr>
<td>Umgebung/Vorhandenem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kundenservice</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tab. 11**: Collection of requirements for software quality from different perspectives
Was kann der Programmierer zur Software-Qualität beitragen?

**Sorgfältige Planung/Vorbereitung:**

- Auf genaue Spezifikation bestehen/Mit Anwendern absprechen
- Genaue Analyse des Problems (Anforderungen, Zielumgebung, ...)
- Auf Korrektheit achten (alle Fälle einbeziehen)
- Auf Vollständigkeit achten (Abgleich mit Spezifikation)
- Einteilung der Funktionen (Voraussicht: Was für Änderungen könnten vorkommen?, Was ist verständlich?)

**Dokumentation:**

- Code kommentieren
- Selbsterklärender Code,
- Redundanz vermeiden
- des Entwurfs, der Systemumgebung = technische Dokumentation
- Handbuch/Hilfe = Benutzerdokumentation

**Tests:**

- Programme vor Übergabe gegenseitig testen

---

*Fig. 3: Summary of the discussion about the programmer’s contribution to software quality*
A.2.2 Experimental Group

Schrittweise Verfeinerung
+ Struktogramme
(revisited)

- Ziel: Überblick -> Korrektheit!
  + Vollständigkeit, Voraussicht

Algorithmus:
1. Zerlege Aufgabe in Teilaufgaben (nimmt dabei an, dass es schon eine Lösung für die Aufgaben gibt)
2. Sind die Teilaufgaben schon implementiert oder einfach genug?
   - ja => implementiere das Programm
   - nein => Zerlege die Teilaufgaben weiter (Schritt 1)

Schrittweise Verfeinerung
+ Struktogramme: Fragen

- Werden die Anforderung vollständig erfüllt?
- Werden die Funktionen korrekt erfüllt?
- Ist die Lösung flexibel?
Beispiel: Intervalle (1)

Algorithmus:

Vergleichen von Intervallen

Es soll geprüft werden, ob zwei Intervalle $A\ (a, b)$ und $B\ (c, d)$ gleich sind, einander enthalten oder sich überschneiden. Es können beliebig viele Paare eingegeben werden, bis das Programm beendet wird.

Beispiel: Intervalle (2)
Beispiel: Intervalle (3)

<table>
<thead>
<tr>
<th>Intervalvergleich IF 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ein Intervall besteht aus zwei ganzen Zahlen Integern $a$, $b$, $c$, $d$</td>
</tr>
<tr>
<td>Eingabe der Intervalle</td>
</tr>
<tr>
<td>Vergleich der Intervalle und Ausgabe</td>
</tr>
<tr>
<td>bis das Programm beendet wird</td>
</tr>
</tbody>
</table>

Beispiel: Intervalle (4)

<table>
<thead>
<tr>
<th>Eingabe der Intervalle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eingabe des Intervalls $A(a, b)$</td>
</tr>
<tr>
<td>Eingabe des Intervalls $B(c, d)$</td>
</tr>
</tbody>
</table>
Beispiel: Intervalle (5)

Beispiel: Intervalle (6)
Beispiel: Intervalle (7)

Quellen

- Der Brockhaus von A-Z in drei Bänden, 2000
B The Sports Club Assignment

The sports club assignment, as described in section 4.2.1, was partly different for the experimental and the control group. We highlight these differences with yellow boxes.

B.1 The Sports Club Assignment for the Experimental Group

The Scenario

A sports club wants to create a statistic about the participation of its members in the different sports types. A programming agency is hired to create a program for that.

It is announced that there will be follow up tasks with extensions of the program.

Description of the Assignment

Each part consists of a programming task and questions about the design. Please submit your complete answers (including the structograms, your thoughts and the program code).

Please do not forget your name in the file names.

Timing

Part 1: 90 min
Part 2 (and 3): 45 min
Submit (at least) part 1 and 2.

Part 1

For every member two numbers are read: Sports type and gender. There is the restriction that every member is active in only one sports type and a maximum of 99 persons in the sports club.

The input values are as follows:

\[
\begin{align*}
0 &= \text{Football} & 0 &= \text{male} \\
1 &= \text{Handball} & 1 &= \text{female} \\
2 &= \text{Volleyball} \\
3 &= \text{Tennis}
\end{align*}
\]

The statistics should be printed this way:

<table>
<thead>
<tr>
<th></th>
<th>male</th>
<th>female</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Football</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>Handball</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>Volleyball</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>Tennis</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>total</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
</tbody>
</table>
1. For the program design, create structograms on at least two levels, i.e. with at least one refinement step. Check the correctness of the design in each step and shortly note down your thoughts.

2. Divide your design into functions. Reflect on how you proceeded for division and describe your strategy.

3. Look at the changeability of your design and note down: What kind of changes do you expect and which ones can be done in only one function?

4. Do you see a better way to divide the functions? Revise your design if necessary. For documenting the design process, explain your changes and your reasons for them.

5. Implement the program. If you deviate from the original design: Document how and why.

Part 2
The sports club has new requirements for the program.

- There can be 999 instead of 99 active people in the sports club.
- Sports types are added:
  - 4 = Badminton
  - 5 = Gymnastics
- In the statistic there should be an additional person group “children”:
  - 2 = Child
- It should be possible to create the statistic for many sports clubs in one program run.

Answer the following questions about the changeability of the program:

1. For each new requirement: In how many locations did you have to change the program in order to fulfill the requirement? Which adjustment was easy, which was difficult?

2. Did you use structograms for the changes? If yes, how?

3. What is your conclusion about the changeability of your original code?

Part 3
Extend the program in a way that it can be used for any kind of club. For that both the number and the names of the activities as well as the number and names of the person groups must be variable.

About this part, answer the same questions as given for part 2.
B.2 The Sports Club Assignment for the Control Group

The Scenario

A sports club wants to create a statistic about the participation of its members in the different sports types. A programming agency is hired to create a program for that.

It is announced that there will be follow up tasks with extensions of the program.

Description of the Assignment

Each part consists of a programming task and questions about the design. Please submit your complete answers (including your thoughts and the program code).

Please do not forget your name in the file names.

Timing

Part 1: 90 min
Part 2 (and 3): 45 min
Submit (at least) part 1 and 2.

Part 1

For every member two numbers are read: Sports type and gender. There is the restriction that every member is active in only one sports type and a maximum of 99 persons in the sports club.

The input values are as follows:

0 = Football 0 = male
1 = Handball 1 = female
2 = Volleyball
3 = Tennis

The statistics should be printed this way:

<table>
<thead>
<tr>
<th></th>
<th>male</th>
<th>female</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Football</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>Handball</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>Volleyball</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>Tennis</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
<tr>
<td>total</td>
<td>99</td>
<td>99</td>
<td>99</td>
</tr>
</tbody>
</table>

1. Plan the program before implementation and explain the plan and your thoughts about it.

2. Implement the program. If you deviate from the original design: Document how and why.
**Part 2**
The sports club has new requirements for the program.

- There can be 999 instead of 99 active people in the sports club.
- Sports types are added:
  - 4 = Badminton
  - 5 = Gymnastics
- In the statistic there should be an additional person group “children”:
  - 2 = Child
- It should be possible to create the statistic for many sports clubs in one program run.

**Answer the following questions about the changeability of the program:**

1. For each new requirement: In how many locations did you have to change the program in order to fulfill the requirement? Which adjustment was easy, which was difficult?
2. Did you plan the changes before coding? If yes, how?
3. What is your conclusion about the changeability of your original code?

**Part 3**
Extend the program in a way that it can be used for any kind of club. For that both the number and the names of the activities as well as the number and names of the person groups must be variable.

About this part, answer the same questions as given for part 2.
C Solutions

C Example Solutions to the Sports Club Assignments

This example solution is constructed by the researcher to illustrate expectations stated more abstractly in section 4.2.2 and as a basis for comparisons with student code fragments as given in section 6.1.

C.1 Part 1

C.1.1 Design

Fig. 4: Very first structogram for sports club assignment, part 1

Refinement level 0 The first structogram (figure 4) shows a very basic design idea for the implementation of the program: The data is initialized, member data is read in and the statistics are output in a table.

Refinement level 1 The next decision we make is what information is needed for each of the blocks and how it needs to be initialized (figure 5). We expect that the club will introduce more sports types and also different ways of grouping the members (i.e. change the number of possible values for each property). Because of that we store the names of the activities and groups in arrays that can be extended if necessary and set the max numbers in constants that are used for the dimensions of the arrays and can be adjusted easily. Another basic question is whether to pass the member information as it is to the output or a set of data that is prepared for the output. We decide for the earlier in order to keep the member data explicit and store this data in an array. Also, we choose the data structure for the statistics data. We have the choice between collecting the information in variables or an array. Using variables, we would have to introduce a new one for each of the expected changes in the possible values for the properties. To keep the program parameterizable we decide to put the information into an array. This array needs to be initialized before use.

On the same level of refinement, read_members (figure 6) prints input instructions and then reads the data for one member until the maximum number of members is reached or the user stops the input. How the input is stopped is postponed for later refinement.
Fig. 5: Structogram on level 1 for sports club assignment, part 1

```
initialize(members, sports, groups)

SPORTS_MAX ← 4
GROUPS_MAX ← 2
MEMBERS_MAX ← 99

sports ← array with sport types names
groups ← array with group names
members ← array with member data
initialize_members(members)
```

Fig. 6: Structogram on level 1 for sports club assignment, part 1

```
read_members(members, sports, groups)

explain_input(sports, groups)

for (counter = 0; counter < MAX_MEMBERS && result != STOP; counter++)

  result ← read_member(members, sports, groups, counter)
```
Output_statistics prepares the output by initializing the statistics array, counting the members and computing the sums, and then prints the table 7. How these steps are done is postponed for further refinement.

Refinement level 2  Initialize_members (figure 8) sets all fields in the member array to -1 as the default value.

Explain_input (figure 9) gives the user instructions how to enter the data.

On the same level, we refine the input of one member (read_member, figure 10) by defining it as the reading in of the sports type and the gender. The return value signals whether the input is finished. The input of the group/second property can be skipped if the end of the input is signaled with the input of the sports type/first property.
**Fig. 9:** Structogram on level 2 for sports club assignment, part 1

**Fig. 10:** Structogram on level 2 for sports club assignment, part 1
Initialize_statistics(statistics) sets all fields in the statistics array to 0.

The computing of the statistics (compute_statistics, figure 12) is done by looping over the member array and counting it in the statistics array by increasing the number in the respective field until an invalid entry is read or the end of the array is reached.

Compute_sums (figure 13) computes the sums for the sums for the horizontal and vertical output in an array each.

The output of the table (output_table, figure 14) consists of the output of the headers, the rows, a line and the footer.

Refinement level 3  Read_input (figure 15) is the same for the sports types and genders: The user is prompted for input. If the input value signals the end of the input, nothing is done, only the information that the input is stopped is returned. Else, the input is checked for validity and if necessary until valid.

The table header consists of an empty cell and the names of the groups plus the heading “sum” (output_header, figure 16).
compute\_sums(statistics, sport\_sums, group\_sums)

\begin{itemize}
\item \texttt{for(group = 0; group \leq group\_max; group++)}
\item \texttt{group\_sums[group] \gets 0}
\item \texttt{for(sport = 0; sport \leq SPORTS\_MAX; sport++)}
\item \texttt{sport\_sums[sport] \gets 0}
\item \texttt{for(group = 0; group \leq GROUPS\_MAX; group++)}
\item \texttt{sport\_sums[sport] \gets sport\_sums[sport] + statistics[sport][group]}
\item \texttt{group\_sums[group] \gets group\_sums[group] + statistics[sport][group]}
\end{itemize}

Fig. 13: Structogram on level 2 for sports club assignment, part 1

output\_table(statistics, sports, groups, sport\_sums, group\_sums)

\begin{itemize}
\item \texttt{output\_headers(groups)}
\item \texttt{output\_rows(statistics, sports, sport\_sums)}
\item \texttt{print\_line}
\item \texttt{output\_footer(group\_sums)}
\end{itemize}

Fig. 14: Structogram on level 2 for sports club assignment, part 1
Fig. 15: Structogram on level 3 for sports club assignment, part 1

Fig. 16: Structogram on level 3 for sports club assignment, part 1
The rows are printed using a loop over the sports with a nested loop for the groups for printing the according statistics data (output_rows, figure 17).

The footer consists of the row heading “sum” and the sums for the groups and the total number of members. The total is computed while looping over the sums for the groups (output_footer, figure 18).

```
output_rows(statistics, sports, sport_sums)

for (sport ← 0; sport < SPORTS_MAX; sport++)
    output sports[sport]

for (group ← 0; group < GROUPS_MAX; group++)
    output statistics[sport][gender]

output sportsum[sport]
```
output_footer(statistics, group_sums)

total ← 0

for (group ← 0; group < GROUPS_MAX; group++)

output group_sums[group]

total ← total + group_sums[group]

output total

Fig. 18: Structogram on level 3 for sports club assignment, part 1
C.1.2 Program Code

/*
 * Sports club, part 1
 * Reads the member data of a sports club and prints out a statistics table
 * author: Miriam Briellmann
 * last change: 14.02.2009
 */

#include <stdio.h>
#include <stdlib.h>

#define SPORTS_MAX 4
#define GROUPS_MAX 2
#define MEMBERS_MAX 99

#define INIT 0
#define OK 1
#define STOP 2

void read_members(int members[][2], char sports[][20], char groups[][20]);
void output_statistics(int members[][2], char sports[][20], char groups[][20]);
void initialize_members(int members[][2]);
void explain_input(char sports[][20], char groups[][20]);
int read_member(int members[][2], char sports[][20], char groups[][20],
                int counter);
void initialize_statistics(int statistics[][GROUPS_MAX]);
void compute_statistics(int members[][2], int statistics[][GROUPS_MAX]);
void compute_sums(int statistics[][GROUPS_MAX], int sport_sums[],
                   int group_sums[]);
void output_table(int statistics[][GROUPS_MAX], char sports[][20],
                  char groups[][20], int sport_sums[], int group_sums[]);
int read_input(char property[], int property_index, char
               property_strings[][20], int property_max, int members[][2],
               int counter);
void output_header(char groups[][20]);
void output_rows(int statistics[][GROUPS_MAX], char sports[][20],
int sports_sums[];
void print_line();
void output_footer(int group_sums[]);

/******************************************************************
* refinement level 0
******************************************************************/

int main() {
    //declaration of shared data, constants as defines
    int members[MEMBERS_MAX][2];
    char sports[SPORTS_MAX][20]={
        "football", "handball", "volleyball", "tennis"};
    char groups[GROUPS_MAX][20]={"male", "female"};
    initialize_members(members);
    read_members(members, sports, groups);
    output_statistics(members, sports, groups);
    system("PAUSE");
}

/******************************************************************
* refinement level 1
******************************************************************/

void read_members(int members[][2], char sports[][20],
                   char groups[][20]) {
    int counter, input=INIT;
    explain_input(sports, groups);
    //read input for each member
    for (counter = 0; counter<MEMBERS_MAX && input!=STOP; counter++) {
        input = read_member(members, sports, groups, counter);
    }
}
void output_statistics(int members[][2], char sports[][20],
char groups[][GROUPS_MAX]) {
    int statistics[SPORTS_MAX][GROUPS_MAX];
    int sport_sums[SPORTS_MAX];
    int group_sums[GROUPS_MAX];
    initialize_statistics(statistics);
    compute_statistics(members, statistics);
    compute_sums(statistics, sport_sums, group_sums);
    output_table(statistics, sports, groups, sport_sums, group_sums);
}

/**
 * initializes the member array to a default, invalid value
 * param int members[][2] the member data of the club
 */
void initialize_members(int members[][2]) {
    int i, j;
    for (i=0; i<MEMBERS_MAX; i++) {
        for (j=0; j<2; j++) {
            members[i][j]=-1;
        }
    }
}

/**
 * prints how the member data should be entered
 * param char sports[][20] the string values for the sports types
 * param char groups[][20] the string values for the person groups
 */
void explain_input(char sports[][20], char groups[][20]) {
    int i;

printf("Please enter the member data as follows: \\
");
printf("Sports types: \\
");
for (i=0; i<SPORTS_MAX; i++) {
    printf("%d: %s \\
", i, sports[i]);
}
printf("Groups: \\
");
for (i=0; i<GROUPS_MAX; i++) {
    printf("%d: %s \\
", i, groups[i]);
}
printf("Stop input with Ctrl+z\\n");
}

/**
 * reads the data for one member
 * param int members[][2] the member data of the club
 * param char sports[][20] the string values for the sports types
 * param char groups[][20] the string values for the person groups
 * return int result signals whether input should be stopped
 */
int read_member(int members[][2], char sports[][20], char groups[][20],
    int counter) {
    int result=INIT, sport=-1, group=-1;
    printf("Please enter data for member %d:\n", counter);
    result = read_input("sport", 0, sports, SPORTS_MAX, members, counter);
    if (result!=STOP)
        result =
        read_input("group", 1, groups, GROUPS_MAX, members, counter);
    return result;
}

/**
 * initializes the statistics array to 0
 * param int statistics[][GROUPS_MAX] the statistics for the combinations
 */
void initialize_statistics(int statistics[][GROUPS_MAX]) {
    int i, j;
    for (i=0; i<SPORTS_MAX; i++) {
        for (j=0; j<GROUPS_MAX; j++) {
            statistics[i][j]=0;
        }
    }
C.1 Part 1

/**
 * compute the statistics as the combination of sports types and person groups
 * for all the members of the club
 * param int members[][2] the member data of the club
 * param int statistics[][GROUPS_MAX] the counts for each combination
 */
void compute_statistics(int members[][2], int statistics[][GROUPS_MAX]) {
    for (i=0; (i<MEMBERS_MAX && members[i][0]!=-1 && members[i][1]!=-1); i++) {
        sport = members[i][0];
        group = members[i][1];
        statistics[sport][group]++;
    }
}

/**
 * compute the sums for the sports types and person groups
 * param int statistics[][GROUPS_MAX] the counts for each combination
 * param int sport_sums[] the number of people for each sports type
 * param int group_sums[] the number of people for each person group
 */
void compute_sums(int statistics[][GROUPS_MAX], int sport_sums[], int group_sums[]) {
    for (group=0; group<GROUPS_MAX; group++)
        group_sums[group]=0;
    for (sport=0; sport<SPORTS_MAX; sport++)
        sport_sums[sport]=0;
    for (i=0; (i<MEMBERS_MAX && members[i][0]!=-1 && members[i][1]!=-1); i++) {
        sport = members[i][0];
        group = members[i][1];
        sport_sums[sport]=sport_sums[sport]+statistics[sport][group];
        group_sums[group]=group_sums[group]+statistics[sport][group];
    }
}

/**
 * prints the statistics table
 * param int statistics[][GROUPS_MAX] the counts for each combination

void output_table(int statistics[][GROUPS_MAX], char sports[][20],
                 char groups[][20], int sport_sums[], int group_sums[])
{
    output_header(groups);
    output_rows(statistics, sports, sport_sums);
    print_line();
    output_footer(group_sums);
}

int read_input(char property_name[], int property_index,
               char property_values[][20], int property_max, int members[][2],
               int counter){
    int result = INIT, read = -1, return_val;
    do {
        printf("\n\%s: ", property_name);
        return_val = scanf("%d", &read);
        fflush(stdin);
        if (return_val==EOF)
            result=STOP;
        else if ((read>=0) && (read<property_max)){
            members[counter][property_index] = read;
            result=OK;
        }
    } while (result!=OK);
} else
    printf("Invalid input. Please repeat input for \%s.\n",
            property_name);
} while ((result!=STOP) && (result!=OK));
return result;
}

/**
 * prints the header of the table
 * param char groups[][20] the string values for the person groups
 */
void output_header(char groups[][20]) {
    int i;
    printf("%12s", "");
    for (i=0; i<GROUPS_MAX; i++) {
        printf("%12s", groups[i]);
    }
    printf("%12s
", "sum");
}

/**
 * prints the rows of the table
 * param int statistics[][GROUPS_MAX] the counts for each combination
 * param char sports[][20] the string values for the sports types
 * param int sport_sums[] the number of people for each sports type
 */
void output_rows(int statistics[][GROUPS_MAX], char sports[][20],
                 int sport_sums[]) {
    int i, j;
    for (i=0; i<SPORTS_MAX; i++) {
        printf("%12s", sports[i]);
        for (j=0; j<GROUPS_MAX; j++) {
            printf("%12d", statistics[i][j]);
        }
        printf("%12d\n", sport_sums[i]);
    }
}

/**
 * prints a line
 */
```c
void print_line() {
    int i, max;
    max = (GROUPS_MAX + 2) * 12;
    for (i=0; i<max; i++) {
        printf("-");
    }
    printf("\n");
}

/**
 * prints the footer of the table including the total number of members
 * param int group_sums[] the number of people for each person group
 */
void output_footer(int group_sums[]) {
    int i, total=0;
    printf("%12s", "sum");
    for (i=0; i<GROUPS_MAX; i++) {
        printf("%12d", group_sums[i]);
        total = total + group_sums[i];
    }
    printf("%12d\n", total);
}
```
C.2 Part 2

C.2.1 Design

For most of the new requirements in part 2 we then only need small adjustments: The names of the sports types and person groups are added to the arrays (1 location each), the numbers of both are changed in the constants (1 location each).

The only major change is the addition of the possibility to generate the statistics table for more than one club in one run which had not been expected. In contrast to the other changes, this is a change in the program structure. Looking at the structograms, it is not very difficult to realize either. What was the main program in the first part, is moved one level down as a function that is called from the new main (see figure 19).

![Structogram on first level for sports club assignment, part 2](image)

Fig. 19: Structogram on first level for sports club assignment, part 2

In the code we highlight the changes that had to be made from part 1 to part 2:

- Add possibility to run for several clubs:
  - move main to function
  - make loop in main

- Change person groups:
  - change max no of person groups (1 location),
  - add string to person groups (1 location),

- Change sports:
  - add string to sports, (1 location)
  - change max no of sports (1 location).

In total 17 lines in 8 locations were changed.
C Solutions

C.2.2 Program Code

1 /*
2  Sports club, part 2
3  Reads the member data of a several sports clubs and for each prints out a
4  statistics table
5  author: Miriam Brielmann
6  last change: 14.02.2009
7 */
8
9 #include <stdio.h>
10 #include <stdlib.h>
11
12 //changed: number of sports
13 #define SPORTS_MAX 6
14 //changed: number of groups
15 #define GROUPS_MAX 3
16 //changed: max number of members
17 #define MEMBERS_MAX 999
18
19 //constants for checking the input
20 #define INIT 0
21 #define OK 1
22 #define STOP 2
23
24 //added:
25 void one_club(int number);
26 void read_members(int members[][2], char sports[][20], char groups[][20]);
27 void output_statistics(int members[][2], char sports[][20], char groups[][20]);
28 void initialize_members(int members[][2]);
29 void explain_input(char sports[][20], char groups[][20]);
30 int read_member(int members[][2], char sports[][20], char groups[][20],
31   int counter);
32 void initialize_statistics(int statistics[][GROUPS_MAX]);
33 void compute_statistics(int members[][2], int statistics[][GROUPS_MAX]);
34 void compute_sums(int statistics[][GROUPS_MAX], int sport_sums[],
35    int group_sums[]);
36 void output_table(int statistics[][GROUPS_MAX], char sports[][20],
37   char groups[][20], int sport_sums[], int group_sums[]);
38 int read_input(char property[], int property_index, char
39    property_strings[][20], int property_max, int members[][2],
40   int counter);
void output_header(char groups[][20]);
void output_rows(int statistics[][GROUPS_MAX], char sports[][20],
    int sports_sums[]);
void print_line();
void output_footer(int group_sums[]);

/**************************************************************/
/* refinement level 0
**************************************************************/

// changed: new main for entering several clubs
int main() {
    int finished = 0, clubno;
    for (clubno = 0; finished!=1; clubno++) {
        fflush(stdin);
        one_club(clubno);
        printf("\nPress any key to continue with entering data for "
            "another club or 1 to quit the program.\n");
        scanf("%d", &finished);
    }
}

/**************************************************************/
/* refinement level 1
**************************************************************/

// changed: made former "main" a function
/**
 * Reads the member data for one club and for each prints out a
 * statistics table
 */
void one_club(int number) {
    // declaration of shared data, constants as defines
    int members[MEMBERS_MAX][2];
    // changed:
    char sports[SPORTS_MAX][20]={
        "football", "handball", "volleyball", "tennis", "badminton",
        "gymnastics"};
    // changed:
    char groups[GROUPS_MAX][20]={"male", "female", "child"};
    initialize_members(members);
read_members(members, sports, groups);
output_statistics(members, sports, groups);
}

/******************************************************************
 * refinement level 2
******************************************************************/
/**
 * reads the member data of a sports club
 * param int members[][2] the member data of the club
 * param char sports[][20] the string values for the sports types
 * param char groups[][20] the string values for the person groups
 */
void read_members(int members[][2], char sports[][20],
                  char groups[][20]) {
    int counter, input=INIT;
    explain_input(sports, groups);
    //read input for each member
    for (counter = 0; counter<MEMBERS_MAX && input!=STOP; counter++) {
        input = read_member(members, sports, groups, counter);
    }
}

/**
 * prints the member data of a sports club
 * param int members[][2] the member data of the club
 * param char sports[][20] the string values for the sports types
 * param char groups[][20] the string values for the person groups
 */
void output_statistics(int members[][2], char sports[][20],
                       char groups[][GROUPS_MAX][20]) {
    int statistics[SPORTS_MAX][GROUPS_MAX];
    int sport_sums[SPORTS_MAX];
    int group_sums[GROUPS_MAX];
    initialize_statistics(statistics);
    compute_statistics(members, statistics);
    compute_sums(statistics, sport_sums, group_sums);
    output_table(statistics, sports, groups, sport_sums, group_sums);
}

```c
123 //**********************************************************************************
124 * refinement level 3
125**********************************************************************************/
126
128 /**
129 * initializes the member array to a default, invalid value
130 * param int members[][2] the member data of the club
131 */
132 void initialize_members(int members[][2]) {
133   int i, j;
134   for (i=0; i<MEMBERS_MAX; i++) {
135     for (j=0; j<2; j++) {
136       members[i][j]=-1;
137     }
138   }
139 }
140
142 /**
143 * prints how the member data should be entered
144 * param char sports[][20] the string values for the sports types
145 * param char groups[][20] the string values for the person groups
146 */
147 void explain_input(char sports[][20], char groups[][20]){
148   int i;
149   printf("Please enter the member data as follows: \n");
150   printf("Sports types: \n");
151   for (i=0; i<SPORTS_MAX; i++) {
152     printf("%d: %s \n", i, sports[i]);
153   }
154   printf("Groups: \n");
155   for (i=0; i<GROUPS_MAX; i++) {
156     printf("%d: %s \n", i, groups[i]);
157   }
158   printf("Stop input with Ctrl+z\n");
159 }
160
162 /**
163 * reads the data for one member
164 * param int members[][2] the member data of the club
165 * param char sports[][20] the string values for the sports types
166 */
```
int read_member(int members[][2], char sports[][20], char groups[][20],
                int counter) {
    int result=INIT, sport=-1, group=-1;
    printf("Please enter data for member %d:\n", counter);
    result = read_input("sport", 0, sports, SPORTS_MAX, members, counter);
    if (result!=STOP)
        result =
        read_input("group", 1, groups, GROUPS_MAX, members, counter);
    return result;
}

void initialize_statistics(int statistics[][GROUPS_MAX]) {
    int i; j;
    for (i=0; i<SPORTS_MAX; i++) {
        for (j=0; j<GROUPS_MAX; j++) {
            statistics[i][j]=0;
        }
    }
}

void compute_statistics(int members[][2], int statistics[][GROUPS_MAX]) {
    int i, sport, group;
    for (i=0; (i<MEMBERS_MAX && members[i][0]!=-1 && members[i][1]!=-1); i++) {
        sport = members[i][0];
        group = members[i][1];
        statistics[sport][group]++;
    }
}
C.2 Part 2

void compute_sums(int statistics[][GROUPS_MAX], int sport_sums[], int group_sums[]) {
    int group, sport;
    for (group=0; group<GROUPS_MAX; group++) {
        group_sums[group]=0;
    }
    for (sport=0; sport<SPORTS_MAX; sport++) {
        sport_sums[sport]=0;
        for (group=0; group<GROUPS_MAX; group++) {
            sport_sums[sport]=sport_sums[sport]+statistics[sport][group];
            group_sums[group]=group_sums[group]+statistics[sport][group];
        }
    }
}

void output_table(int statistics[][GROUPS_MAX], char sports[][20], char groups[][20], int sport_sums[], int group_sums[]) {
    output_header(groups);
    output_rows(statistics, sports, sport_sums);
    print_line();
    output_footer(group_sums);
}

/********************
* refinement level 4

**************************************************************************

/**
 * reads the data for one property of a member and checks it until valid data
 * is entered or input is terminated by the user
 * param char property_name[] the name of the property
 * param int property_index the index of the property in the array for saving
 * param char property_values[][20] the string values for the property
 * param int property_max the number of values for the property
 * param int members[][2] the member data of the club
 * return int result signals whether input should be stopped
 */

int read_input(char property_name[], int property_index,
               char property_values[][20], int property_max, int members[][2],
               int counter){
    int result = INIT, read = -1, return_val;
    do {
        printf("\n%s: ", property_name);
        return_val = scanf("%d", &read);
        fflush(stdin);
        if (return_val==EOF)
            result=STOP;
        else if ((read>=0) && (read<property_max)){
            members[counter][property_index] = read;
            result=OK;
        } else
            printf("\nInvalid input. Please repeat input for %s.",
                   property_name);
    } while ((result!=STOP) && (result!=OK));
    return result;
}

/**
 * prints the header of the table
 * param char groups[][20] the string values for the person groups
 */

void output_header(char groups[][20]) {
    int i;
    printf("%12s", "");
    for (i=0; i<GROUPS_MAX; i++) {
printf("%12s", groups[i]);
}
printf("%12s\n", "sum");
}

/**
 * prints the rows of the table
 * param int statistics[][GROUPS_MAX] the counts for each combination
 * param char sports[][20] the string values for the sports types
 * param int sport_sums[] the number of people for each sports type
 */
void output_rows(int statistics[][GROUPS_MAX], char sports[][20],
                 int sport_sums[]) {
    int i, j;
    for (i=0; i<SPORTS_MAX; i++) {
        printf("%12s", sports[i]);
        for (j=0; j<GROUPS_MAX; j++) {
            printf("%12d", statistics[i][j]);
        }
        printf("%12d\n", sport_sums[i]);
    }
}

/**
 * prints a line
 */
void print_line() {
    int i, max;
    max = (GROUPS_MAX + 2) * 12;
    for (i=0; i<max; i++) {
        printf("-");
    }
    printf("\n");
}

/**
 * prints the footer of the table including the total number of members
 * param int group_sums[] the number of people for each person group
 */
void output_footer(int group_sums[]) {
    int i, total=0;
for (i=0; i<GROUPS_MAX; i++) {
    printf("%12d", group_sums[i]);
    total = total + group_sums[i];
}
printf("%12d
", total);

C.3 Part 3

C.3.1 Design

For making the program usable for all kinds of clubs, we change the data initialization part (initialize, figure 20). We keep the upper limits for the activities and the groups in constants as we need them for the sizes of the arrays. However, the names of the arrays are filled by user input. The actual number of entered activities and groups is used instead of the fixed maximum where necessary, i.e. for as upper limit for the loops and input check. For that, we change the parameters of 9 functions. For completeness, we also replace all occurrences of “sports” by “activities”.

Read_strings is used for reading the names of the properties. It works the same way for the activities and groups (figure 21).
Fig. 21: Structogram on level 2 for sports club assignment, part 3
C.3.2 Program Code

/*
 * Sports club, part 3
 * Reads the member data of several clubs and for each prints out a
 * statistics table
 * author: Miriam Brielmann
 * last change: 14.02.2009
 */

#include <stdio.h>
#include <stdlib.h>

//changed: number of activities
#define ACTIVITIES_MAX 10

//changed: number of groups
#define GROUPS_MAX 4

//max number of members
#define MEMBERS_MAX 999

//constants for checking the input
#define INIT 0
#define OK 1
#define STOP 2

void one_club(int number);

//added:
int read_strings(char property_name[], char property_strings[][20], int max);

//changed parameters:
void read_members(int members[][2], char activities[][20], char groups[][20],
                   int no_activities, int no_groups);

//changed parameters:
void output_statistics(int members[][2], char activities[][20],
                        char groups[][20], int no_activities, int no_groups);

void initialize_members(int members[][2]);

//changed parameters:
void explain_input(char activities[][20], char groups[][20],
                   int no_activities, int no_groups);
int read_member(int members[][2], char activities[][20], char groups[][20],
                int counter, int no_activities, int no_groups);

void initialize_statistics(int statistics[][GROUPS_MAX]);

void compute_statistics(int members[][2], int statistics[][GROUPS_MAX]);
//changed parameters:
void compute_sums(int statistics[][GROUPS_MAX], int activity_sums[],
    int group_sums[], int no_activities, int no_groups);

//changed parameters:
void output_table(int statistics[][GROUPS_MAX], char activities[][20],
    char groups[][20], int activity_sums[], int group_sums[],
    int no_activities, int no_groups);
int read_input(char property[], int property_index, char
    property_strings[][20], int property_max, int members[][2],
    int counter);

//changed parameters:
void output_header(char groups[][20], int no_groups);

//changed parameters:
void output_rows(int statistics[][GROUPS_MAX], char activities[][20],
    int activities_sums[], int no_activities, int no_groups);

//changed parameters:
void print_line(int no_groups);

//changed parameters:
void output_footer(int group_sums[], int no_groups);

/******************************************************************
* refinement level 0
******************************************************************/
int main() {
    int finished = 0, clubno;
    for (clubno = 0; finished!=1; clubno++) {
        fflush(stdin);
        one_club(clubno);
        printf("\nPress any key to continue with entering data for "
            "another club or 1 to quit the program.\n");
        scanf("%d", &finished);
    }
}

/******************************************************************
* refinement level 1
******************************************************************/

/**
 * Reads the member data for one club and for each prints out a
 * statistics table
void one_club(int number) {
    //declaration of shared data, constants as defines
    int members[MEMBERS_MAX][2];
    //changed:
    char activities[ACTIVITIES_MAX][20];
    //changed:
    char groups[GROUPS_MAX][20];
    //added:
    int no_activities, no_groups;
    no_activities = read_strings("activity", activities, ACTIVITIES_MAX);
    no_groups = read_strings("group", groups, GROUPS_MAX);
    initialize_members(members);
    printf("Enter data for club no %d: \
", number);
    read_members(members, activities, groups, no_activities, no_groups);
    output_statistics(members, activities, groups, no_activities, no_groups);
}

int read_strings(char property_name[], char property_strings[][20], int max){
    int counter=0, return_val=0;
    printf("Enter information for property \\
", property_name);
    do{
        printf("%d:", counter);
        return_val = scanf("%s", property_strings[counter]);
        if (return_val!=EOF) counter++;
    }while((counter<max) || (return_val!=EOF));
    return counter;
}
/**
 * reads the member data of a club
 * param int members[][2] the member data of the club
 * param char activities[][20] the string values for the activities
 * param char groups[][20] the string values for the person groups
 * param int no_activities the number of activities
 * param int no_groups the number of person groups
 */

void read_members(int members[][2], char activities[][20],
char groups[][GROUPS_MAX], int no_activities, int no_groups) {
    int counter, input = INIT;
    explain_input(activities, groups, no_activities, no_groups);
    // read input for each member
    for (counter = 0; counter < MEMBERS_MAX && input != STOP; counter++) {
        input = read_member(members, activities, groups, counter,
                             no_activities, no_groups);
    }
}

/**
 * prints the member data of a club
 * param int members[][2] the member data of the club
 * param char activities[][20] the string values for the activities
 * param char groups[][20] the string values for the person groups
 * param int no_activities the number of activities
 * param int no_groups the number of person groups
 */

void output_statistics(int members[][2], char activities[][20],
char groups[][GROUPS_MAX], int no_activities, int no_groups) {
    char groups[][GROUPS_MAX];
    int statistics[ACTIVITIES_MAX][GROUPS_MAX];
    int activity_sums[ACTIVITIES_MAX];
    int group_sums[GROUPS_MAX];
    initialize_statistics(statistics);
    compute_statistics(members, statistics);
    compute_sums(statistics, activity_sums, group_sums, no_activities,
                 no_groups);
    output_table(statistics, activities, groups, activity_sums, group_sums,
                 no_activities, no_groups);
}
refinement level 3

initialize members(int members[] [2]) {
    int i, j;
    for (i=0; i<MEMBERS_MAX; i++) {
        for (j=0; j<2; j++) {
            members[i][j]=-1;
        }
    }
}

explain input(char activities[] [20], char groups[] [20], int no_activities, int no_groups) {
    int i;
    printf("Please enter the member data as follows: \n");
    printf("Activity: \n");
    for (i=0; i<no_activities; i++) {
        printf("%d: %s \n", i, activities[i]);
    }
    printf("Groups: \n");
    for (i=0; i<no_groups; i++) {
        printf("%d: %s \n", i, groups[i]);
    }
    printf("Stop input with Ctrl+z\n");
}
/**
 * reads the data for one member
 * param int members[][2] the member data of the club
 * param char activities[][20] the string values for the activities
 * param char groups[][20] the string values for the person groups
 * param int counter the number of the member
 * param int no_activities the number of activities
 * param int no_groups the number of person groups
 * return int result signals whether input should be stopped
 */

int read_member(int members[][2], char activities[ACTIVITIES_MAX][20],
char groups[GROUPS_MAX][20], int counter, int no_activities,
int no_groups) {
    int result=INIT;
    printf("Please enter data for member %d:\n", counter);
    result = read_input("activity", 0, activities, no_activities,
members, counter);
    if (result!=STOP)
        result =
        read_input("group", 1, groups, no_groups, members, counter);
    return result;
}

/**
 * initializes the statistics array to 0
 * param int statistics[][GROUPS_MAX] the statistics for the combinations
 */

void initialize_statistics(int statistics[][GROUPS_MAX]) {
    int i, j;
    for (i=0; i<ACTIVITIES_MAX; i++) {
        for (j=0; j<GROUPS_MAX; j++) {
            statistics[i][j]=0;
        }
    }
}

/**
 * compute the statistics as the combination of activities and person groups
 * for all the members of the club
 * param int members[][2] the member data of the club
 * param int statistics[][GROUPS_MAX] the counts for each combination
 */
```c
void compute_statistics(int members[][2], int statistics[][GROUPS_MAX]) {
    int i, activity, group;
    for (i=0; (i<MEMBERS_MAX && members[i][0]!=-1 && members[i][1]!=-1); i++) {
        activity = members[i][0];
        group = members[i][1];
        statistics[activity][group]++;
    }
}

/**
 * compute the sums for the activities and person groups
 * param int statistics[][GROUPS_MAX] the counts for each combination
 * param int activity_sums[] the number of people for each activity
 * param int group_sums[] the number of people for each person group
 * param int no_activities the number of activities
 * param int no_groups the number of person groups
 */

void compute_sums(int statistics[][GROUPS_MAX], int activity_sums[], int group_sums[], int no_activities, int no_groups) {
    int group, activity;
    for (group=0; group<no_groups; group++) {
        group_sums[group]=0;
    }
    for (activity=0; activity<no_activities; activity++) {
        activity_sums[activity]=0;
        for (group=0; group<no_groups; group++) {
            activity_sums[activity]=activity_sums[activity]+statistics[activity][group];
            group_sums[group]=group_sums[group]+statistics[activity][group];
        }
    }
}

/**
 * prints the statistics table
 * param int statistics[][GROUPS_MAX] the counts for each combination
 * param char activities[][20] the string values for the activities
 * param char groups[][20] the string values for the person groups
 * param int activity_sums[] the number of people for each activity
 * param int group_sums[] the number of people for each person group
 */
```
void output_table(int statistics[][GROUPS_MAX], char activities[][20],
char groups[][20], int activity_sums[], int group_sums[],
int no_activities, int no_groups) {
output_header(groups, no_groups);
output_rows(statistics, activities, activity_sums, no_activities,
no_groups);
print_line(no_groups);
output_footer(group_sums, no_groups);
}

int read_input(char property_name[], int property_index,
char property_values[][20], int property_max, int members[][2],
int counter){
int result = INIT, read = -1, return_val;
do {
printf("\n%", property_name);
return_val = scanf("%d", &read);
fflush(stdin);
if (return_val==EOF)
result=STOP;
else if ((read>=0) && (read<property_max)){
members[counter][property_index] = read;
result=OK;
} else
    printf("Invalid input. Please repeat input for \%s.\n",
        property_name);
} while ((result!=STOP) && (result!=OK));
return result;
}

/**
 * prints the header of the table
 * param char groups[][20] the string values for the person groups
 * param int no_groups the number of person groups
 */
void output_header(char groups[][20], int no_groups) {
    int i;
    printf("%12s", "");
    for (i=0; i<no_groups; i++) {
        printf("%12s", groups[i]);
    }
    printf("%12s\n", "sum");
}

/**
 * prints the rows of the table
 * param int statistics[][GROUPS_MAX] the counts for each combination
 * param char activities[][20] the string values for the activities
 * param int activity_sums[] the number of people for each activity
 * param int no_activities the number of activities
 * param int no_groups the number of person groups
 */
void output_rows(int statistics[][GROUPS_MAX], char activities[][20],
                 int activity_sums[], int no_activities, int no_groups) {
    int i, j;
    for (i=0; i<no_activities; i++) {
        printf("%12s", activities[i]);
        for (j=0; j<no_groups; j++) {
            printf("%12d", statistics[i][j]);
        }
        printf("%12d\n", activity_sums[i]);
    }
}
/**
 * prints a line
 * param int no_groups the number of person groups
 */
void print_line(int no_groups) {
    int i, max;
    max = (no_groups + 2) * 12;
    for (i=0; i<max; i++) {
        printf("-");
    }
    printf("\n");
}

/**
 * prints the footer of the table including the total number of members
 * param int group_sums[] the number of people for each person group
 * param int no_groups the number of person groups
 */
void output_footer(int group_sums[], int no_groups) {
    int i, total=0;
    printf("%12s", "sum");
    for (i=0; i<no_groups; i++) {
        printf("%12d", group_sums[i]);
        total = total + group_sums[i];
    }
    printf("%12d\n", total);
}
D References


