Pseudocode, the Pseudocode Programming Process, and Alternatives to the PPP

Noah Doersing

noah.doersing@student.uni-tuebingen.de

Abstract. Writing pseudocode before source code eases the development process, helping to fix certain kinds of errors before any source code is written and providing easily maintainable documentation. Studies show that pseudocode is superior to visual approaches for routine creation in most programming environments, while visualizations are more effective when used as a teaching aid. The Pseudocode Programming Process (PPP) for routine creation is introduced in detail and compared to some well-known alternatives.

Keywords: pseudocode, ppp, programming, routine creation, software engineering, code, flowcharts, prototyping, refactoring, test-driven development, design by contract

1 Introduction

When writing software, programmers will reach a point when the class structure has been designed and the major routines inside each class have been defined, but no code has been written yet. Taking the next step - writing the routines from scratch in an elegant, efficient and maintainable way - is a near-impossible task without a systematic approach.

One of these approaches, and the one this paper will discuss in detail, is called the Pseudocode Programming Process (PPP). It utilizes pseudocode to enable iterative routine creation, providing guidelines and simple, straightforward steps for going from a high-level description to low-level source code, thus aiding the programmer in efficient code creation. Further, it allows the detection of some kinds of errors before any source code is written. As an additional benefit, accurate and complete documentation is created with very little additional effort when following the steps outlined here.

I will first describe what constitutes good pseudocode in this context, giving positive and negative examples. Then I will give a more in-depth example and explanation of the PPP, highlighting some of the points a programmer should keep in mind while creating a routine. Afterwards, I will mention and interpret results of some studies investigating the applications of pseudocode, before discussing the advantages and disadvantages of the PPP as well as some alternative approaches and how they compare to pseudocode. Finally, I will summarize my findings and mention some items for consideration in future research.

1.1 Background

As this paper is discussing a software development tool, experiences in programming and software development methodologies are beneficial. The examples will be written in C and Java, however no deep knowledge of any specific programming language is required.

2 Contribution

2.1 Good Pseudocode

In this section, outline some criteria regarding the quality of pseudocode will be outlined, and what characterizes good pseudocode will be defined.

Pseudocode in the context of this paper consists of precise, natural-language descriptions of specific actions. This sets it apart from other kinds of pseudocode commonly used in algorithm papers or classes, where pseudocode is understood as a way of writing down programs at a relatively low level without adhering to the specific rules of any programming language. Of course, when implementing a formula or anything that is more conventionally expressed in a textual representation other than natural language, this guideline loses some validity.

Good pseudocode avoids code-like statements: it tries to stay at a higher level than source code in order to increase the efficiency during the design phase and to avoid getting restricted by specific limitations or features of the target programming language. This allows the programmer to capture the intent of an action inside a routine: Pseudocode should describe what a design is doing, not how exactly it is going to be implemented.

However the level of the pseudocode should be low enough to enable the programmer to refine it into source code very quickly (almost automatically).

Fig. 1 provides an example of pseudocode following these guidelines. Its purpose (creating a dialog box and adding it to some data structure) can be easily guessed without further knowledge about any implementation details.

```
Keep track of current number of resources in use
If another resource is available
    Allocate a dialog box structure
    If a dialog box structure could be allocated
        Note that one more resource is in use
        Initialize the resource
        Store the resource number at the
          location provided by the caller
    Endif
Endif
Return true if a new resource was created;
  else return false
```

Fig. 1. Example: good pseudocode

```
increment resource number by 1
allocate a dlg struct using malloc
if malloc() returns NULL then return 1
invoke OSrsrc_init to initialize a resource
  for the operating system
*hRsrcPtr = resource number
return 0
```

Fig. 2. Example: bad pseudocode

On the other hand, as shown in Fig. 2, failing to follow the guidelines produces very hard-to-read pseudocode, in this case with many superfluous low-level implementation details (C-specific terms such as `dlg struct`, `malloc` or `OSrsrc_init` and conventions such as `0` meaning `true`) leaking through.

Two points that will be examined in more detail in the "Discussion" section are that pseudocode allows the programmer to quickly evaluate multiple different approaches and that writing pseudocode is an iterative process, which will also become apparent in the following example.

2.2 Example: Constructing a routine with the PPP

This section will give an example of going through the PPP workflow, highlighting some points to keep an eye on during the routine creation process.

The necessary steps are, in order, the design of the routine, writing pseudocode, translating the pseudocode into source code, and checking and testing the routine, whereupon some of the steps may be repeated multiple times depending on design and engineering challenges.

Design Suppose that after designing the class structure and outlining the main routines, the programmer wants to write a routine that reads the grades for students taking a specific class ("SCT") in a given year from a database, computes the final grade for each student as a weighted average of the grades, and returns a list of student names and final grades.

After writing an informal spec of the routine as above, the programmer should first check the prerequisites: they need to make sure that the routine is well defined, necessary, and that it fits cleanly into the class design. Unless the informal spec is already detailed enough, the problem solved by the routine has to be clearly defined, along with inputs, outputs, assumed preconditions and assured postconditions. In the aforementioned example, the input is a year, there are no guaranteed preconditions, and the output is either a list of names and grades or a boolean indicating failure.

Other important tasks in this stage include naming the routine, settling on a strategy for testing the routine, researching already available functionality in order not to write duplicate code (as well as searching for prior solutions to similar problems), considering how to handle errors (in the example: how to handle years in which the class wasn't offered and how to handle database connection errors) and, if necessary, how to write the routine to run as efficiently as possible.

Pseudocode After the previous steps have been mentally executed, the programmer first writes a short high-level description of the routine (Fig. 3), which might later be used as the header comment of the routine for the API documentation (e.g. using Javadoc), in order to make sure they understand it before writing the initial, still high-level, iteration of pseudocode (Fig. 4). If writing a concise high-level description seems like a hard task, it is generally beneficial to review the routine design again and to consider splitting it up into multiple smaller routines.

```
This routine returns a list of all students
  taking the SCT course in the given year,
  along with their final grades.
```

Fig. 3. Example: high-level routine description

```
validate the year
retrieve a list of students from the database
compute the final grades
return the names and grades as a list
```

Fig. 4. Example: high-level pseudocode

Now the pseudocode is iteratively refined and checked until it is detailed enough to be translated into source code without a lot of effort (Fig. 5). At this stage, the programmer can easily and quickly evaluate multiple approaches to solve particularly tricky parts of the problem.

```
if the year is valid
    connect to the database and retrieve info about all students in the given year
    keep a list of the students and their final grades
    for all students
        calculate a weighted average of the three grades
        add their name and final grade to the list
    endfor
endif
return the list; or return false if no list could be generated for the given year
```

Fig. 5. Example: mid-level pseudocode

Before continuing on to the coding stage, the programmer should take a step back and check the pseudocode, trying to discover high-level mistakes and making sure to understand all components first. This can also easily be performed by a coworker who might bring in a different perspective.

Code Once the pseudocode is written and checked, the programmer writes the routine declaration and adds the high-level header comment. Then the pseudocode is copied into the body of the routine and turned into inline comments. This serves as a blueprint and framework for code creation, where the source code below each pseudocode comment is iteratively filled in (Fig. 6).

As one can see in the example, each line of pseudocode has resulted in one or multiple lines of source code. Normally, 2 to 10 lines of source code should be generated out of a single pseudocode statement, but this example is near the low end of that range due to its simplicity.

Check and Test After finishing the implementation of a routine, the programmer should verify that the implementation is elegant, correct, and that it satisfies the original spec. A few points to keep an eye on during this stage are outlined below.

Check whether to move some of the code into a separate routine. This might be necessary when a piece of code is used in multiple places in the routine (or

```
/**
 * This routine returns a list of all students taking the SCT course in
 * the given year, along with their final grades.
 */

public List<Pair<String, Float>> listFinalGrades(int year) {
    // if the year is valid
    if (validateYear(year)) {
        // connect to the database and retrieve info about all students in the given year
        [...] Student[] students = [...]

        // keep a list of the students and their final grades
        List list = new ArrayList<Pair<String, Float>>();

        // for all students
        for (Student student : students) {
            // calculate a weighted average of the three grades
            float finalGrade = 0.4 * student.presentationGrade
                             + 0.4 * student.termpaperGrade
                             + 0.2 * student.reviewGrade;

            // add their name and final grade to the list
            list.add(new Pair<String, Float>(student.name, finalGrade));
        }

        // return the list; or return false if no list could be generated for the given year
        return list;
    }
    return false;
}
```

Fig. 6. Example: (shortened) routine as a result of the PPP

program) or when it might come in useful in future routines as well. Additionally, moving long blocks of code into a new routine makes the original routine more concise. In these cases, it should be easy to determine the name of the new routine from the pseudocode. Apply the PPP recursively to make sure the resulting routine is well-designed and maintainable.

Mentally execute each path the routine can take based on different inputs or preconditions, and check for errors. This also serves to deepen the programmer's understanding of the code, making the following checking and testing steps easier, and can be performed by or with a coworker.

After finishing this review, compile the routine. It is not advisable to do this earlier as it tends to make the programmer focus prematurely on fixing small mistakes like undeclared variables, mistyped variable names or missing line terminators, leading to the programmer potentially overlooking higher-level errors. Setting the compiler's warning level to the highest setting allows the programmer to catch many subtle errors and inaccuracies (which can be addressed more easily now as they will not be buried in more major errors) and to improve their coding style in order to avoid repeating the same small mistakes over and over again.

After the first successful compilation, the programmer should debug and test the routines in accordance with well-known guidelines. If many errors surface in this part of the development process, it may be advisable to consider adapting the pseudocode and rewriting the routine from scratch.

The penultimate part of this process is to go through the routine again, cleaning up "leftovers". This includes revisiting the inputs, making sure all parameters of the routine are used, refactoring variable names to improve descriptiveness and consistency, searching for off-by-one errors and potentially infinite loops, making sure white space is used correctly, and various other small improvements.

Finally, the pseudocode (now in the form of comments) is inspected and adapted to more accurately reflect the source code it describes. Lines of pseudocode that have been made redundant (e.g. due to variable naming or the structure of the code) are removed at this stage, making the routine easier to read and maintain in the future.

3 Evaluation

In this section, I will summarize, quote, and interpret some studies that have been conducted to investigate various aspects of pseudocode, largely comparing pseudocode to other ways of representing a program before the coding stage.

One study mentions that text has a higher information density than equivalent visualizations [2], suggesting that pseudocode may be more readable than visual representations of the same routine.

A widely-cited study by D. A. Scanlan [3] finds that "students overwhelmingly preferred structured flowcharts over pseudocode for comprehending the algorithms presented", with the "results strongly [indicating] that structured flowcharts do indeed aid algorithm comprehension. A large difference was found even for the simplest algorithm." However this study only evaluates pseudocode as a teaching aid and not as a software engineering tool, rendering the interesting results largely irrelevant here.

Another study comparing flowcharts and Program Design Languages (PDL, which resemble pseudocode as defined in the context of this paper) [4] finds that "the use of a PDL by a software designer, for the development and description of a detailed program design, produced better results than did the use of flowcharts. Specifically, the designs appeared to be of significantly better quality, involving more algorithmic or procedural detail, than those produced using flowcharts. [...] When equivalent, highly readable designs were presented to subjects in both PDL and flowchart form, no pattern of short-term or long-term differences in comprehension of the design was observed. No significant differences were detected in the quality or other properties of programs written as implementations of the designs." "Subjective ratings indicated a mild preference for PDLs and indicated several specific criteria on which students found PDLs preferable to flowcharts."

Finally, M. K. Hayden finds [6] that pseudocode performs better than algorithmic state machine charts (ASM charts, closely related to flowcharts) as a development aid for programming in a text-based language, while ASM charts are superior to pseudocode if the target programming environment is a "diagrammatic computer interface language", suggesting that design tools should be closely matched to implementation tools.

Summarizing these studies, it seems that pseudocode is more effective than visual approaches during conventional software design, but visualizations are still beneficial in other areas, especially in education.

4 Discussion

After introducing the PPP and evaluating it in the previous sections, we can draw some conclusions about the advantages and disadvantages of using pseudocode in the software development process.

4.1 Benefits of the PPP

The PPP allows the programmer to review a complex routine design very early in the development process (before writing any source code), providing an efficient way of verifying that a design works and catching potential mid-level errors. This reduces the need to review the code itself. If the pseudocode is kept (e.g. in the form of inline comments) and any changes or additions are made to the pseudocode first, this benefit is also carried over to the maintenance and modification stage of software development, helping to reduce the number of errors accumulating over time.

By using the high-level description of the routine as a header comment for the API documentation and by keeping the non-redundant lines of pseudocode around as inline comments, commenting effort is significantly reduced. This also serves as easily maintainable documentation: source code and comments are in the same place, which makes it easier and more natural to keep the program and the documentation in "sync" with each other.

The PPP supports the idea of iterative refinement: A high-level description is refined to pseudocode, which in turn is refined into source code. This provides a simple path for the programmer to follow for routine creation and helps them avoid getting stuck on some problem. As a result of this approach, high-level errors are caught at the design stage, mid-level errors at the pseudocode stage, and low-level errors at the coding stage.

Another benefit of writing pseudocode before source code is that pseudocode takes much less time to write. This enables the programmer to evaluate multiple approaches or variations on an approach before settling on one to implement, which ideally leads to implementing only the most efficient or maintainable approach.

4.2 Disadvantages of the PPP

As any software development approach, the PPP has some downsides.

Writing pseudocode before source code, as well as following the other guidelines of the PPP, adds some overhead to the software development process. This eases the construction of complex routines, but may also result in increasing the time spent writing simpler components of a program.

Further, it is not always clear when to stop writing pseudocode and start writing source code instead. The programmer might spend some time writing unnecessarily detailed pseudocode without noticing.

Keeping this pseudocode as inline comments after coding the routine often results in some redundancy or "overdocumentation". Actively removing redundant comments serves to remedy this.

It's worth pointing out that automated tests as well as formal interface specifications including preconditions, postconditions and invariants, which are essential to the development of some types of software, are not a central part of the PPP, however it is possible to adapt it to more strongly focus on contracts between routines and tests by combining it with the following alternatives.

4.3 Alternatives to the PPP

In this section, a number of alternative software development approaches will be discussed. None of them are equivalent to the PPP and as such cannot fully replace pseudocode, but they can be combined with or without the PPP, with their application also leading to elegant routines. At the end of this section, the alternatives approaches will be compared to the PPP and potential shortcomings will be pointed out.

Flowcharts. This approach provides a visual way of defining/designing the control flow inside a routine. Due to their visual nature, flowcharts are easier to use on paper or a whiteboard than a computer, but they can get relatively messy quickly if the design is changed (e.g. as a result of iterative refinement). For large or complex routines, a visual representation can be less readable than pseudocode [2]. This tool also works better for low-level algorithm design, and as such can not be used to effectively create most routines. Much like pseudocode, floacharts are also applicable as a teaching aid, perhaps more effectively so [3].

(Rapid) Prototyping. Before implementation in the target programming language, a prototype is constructed in a higher-level language or with a specialized tool. In addition to guiding the final implementation of the routine and catching errors, this enables the programmer to evaluate the effectiveness of the approach and allows for preliminary (user) testing where required. Two types of prototypes are distinguished between[1]: Horizontal prototypes serve to display a wide range of features without implementing any in detail, while vertical prototypes are focused on near-complete implementation of a specific aspect of a system. For routine creation, vertical prototypes are most commonly used.

[1] `http://www.usabilityfirst.com/glossary/horizontal-and-vertical-prototypes/`

For example, suppose a programmer wants to write an image comparison program in C. They might first write a prototype in Matlab, which is a higher-level language with significant image processing functionality already built-in, to verify and improve their approach and make sure it works on many kinds of images. After rapidly iterating and arriving at a well-performing algorithm, the programmer reimplements the program in C for maximum processing speed in an embedded system.

"Build One To Throw Away". Approximates prototyping in the target programming language. After a quick preliminary implementation, another version is implemented in the same programming language, which is expected to be more well-designed and contain fewer errors than the first one.

The following three approaches are presented in Code Complete [1]. They provide less complete routine creation workflows compared to the approaches that have already been introduced, as will be discussed in the following section, but they are still worth mentioning as potential additions to an organization's software development process.

Refactoring. After routine creation, the programmer locates and eliminates "code smells", e.g. long routines, deeply nested loops, inconsistent or inaccurate naming, and duplicated code. As a result of making sure that the source code follows such best practices, code quality and maintainability is improved [5].

Test-driven development. Before writing any source code directly belonging to the routine in construction, an automated test case ("unit test") is written. Then source code is added to the routine until it passes the test case. If necessary, another test case is written, followed by the addition of enough source code for the routine to pass the test. This process is repeated until the routine is complete.

Design by Contract. Routines are defined based on their preconditions (assertions regarding the state of the program before the routine is called and any inputs to the routines) and postconditions (assertions regarding the state after termination of the routine, as well as its outputs).

4.4 Comparison of alternative approaches to the PPP

Only Flowcharts, Prototyping and "Build One To Throw Away" can be used to ease the routine creation process by supporting iterative refinement.

Refactoring can be applied in any case to catch any errors or "code smells" after the initial implementation in the target programming language. Test-driven development divides the routine creation into multiple passes over the routine, where in each step some additional functionality is added, and as such eases development, but can also result in messy patched-up code if little attention to code quality is paid during the process (and it is not combined with a final refactoring step). Design by contract barely qualifies, as it does not aid routine creation in any way other than making sure inputs and outputs are well-defined, and as a result is merely useful when combined with other approaches.

While all approaches are in principle applicable in any programming language, different programming languages lend themselves to different approaches and vice versa. For example, design by contract may be more necessary in high-level languages with dynamic type systems, while detailed pseudocode might be superfluous in very high-level languages.

All of the alternatives to the PPP that I have introduced in this section do not provide some of the benefits of the PPP: They do not "automatically" result in inline documentation, and iterative refinement during routine creation, which is very beneficial to coming up with an elegant design, is sometimes problematic.

5 Conclusion

We've seen the PPP as a versatile approach to routine creation, which is one of the most important aspects of software development. In doing so, we've discussed what discerns good pseudocode from bad pseudocode, and how pseudocode compares to more visual approaches in multiple studies. In addition, we've explored some alternatives to using pseudocode, and talked about how they compare.
Due to the aforementioned benefits of the PPP, its simple adaptability and because most alternatives don't allow for iterative refinement or near-automatic inline documentation generation, I conclude that the PPP is an essential tool for efficient software development.

An interesting matter to investigate in future research might be potential improvements to the PPP for better suitability for software development in higher-level languages, perhaps through combining it with prototyping or test-driven development. Even before that, more studies comparing the PPP to other non-visual routine creation approaches might be valuable in order to evaluate which parts of which approach make the largest difference for creating efficient, maintainable, well-documented code.

References

1. McConnell, S.: Code Complete, 2nd Edition, 216–234 (2004)
2. Stankovic, N., Kranzelmller, D., Zhang, K.: The PCG: An Empirical Study. Journal of Visual Languages and Computing 12, 203–216 (2001)
3. Scanlan, D. A.: Structured flowcharts outperform pseudocode: an experimental comparison, IEEE Software 6 (5), 2836 (1989)
4. Ramsey, H. R., Atwood, M. E., van Doren, J. R.: Flowcharts Versus Program Design Languages: An Experimental Comparison, Communications of the ACM 26 (6) 445–449 (1983)
5. Fowler, M.: Refactoring: Improving the Design of Existing Code (1999)
6. Hayden, M. K., Olfman, L., Gray, P., Ahituv, N.: An Experimental Investigation of Visual Enhancements for Programming Environments, Journal of Information Systems, Fall 1997, 19–26 (1997)