

Why the Unix Philosophy still matters

markus schnalke <meillo@marmaro.de>

ABSTRACT

This paper explains the importance of the Unix Philosophy for software design. Today, few software designers are aware of these concepts, and thus a lot of modern software is more limited than necessary and makes less use of software leverage than possible. Knowing and following the guidelines of the Unix Philosophy makes software more valuable.

1. Introduction

The Unix Philosophy is the essence of how the Unix operating system, especially its toolchest, was designed. It is not a limited set of fixed rules, but a loose set of guidelines which tell how to write software that suites Unix well. Actually, the Unix Philosophy describes what is common in typical Unix software. The Wikipedia has an accurate definition [23]:

The Unix philosophy is a set of cultural norms and philosophical approaches to developing software based on the experience of leading developers of the Unix operating system.

As there is no single definition of the Unix Philosophy, several people have stated their view on what it comprises. Best known are:

- Doug McIlroy's summary: "Write programs that do one thing and do it well." [11]
- Mike Gancarz' book "The UNIX Philosophy" [8].
- Eric S. Raymond's book "The Art of UNIX Programming" [15].

These different views on the Unix Philosophy have much in common. Especially, the main concepts are similar in all of them. McIlroy's definition can surely be called the core of the Unix Philosophy, but the fundamental idea behind it all is "small is beautiful".

The Unix Philosophy explains how to design good software for Unix. Many concepts described here are based on Unix facilities. Other operating systems may not offer such facilities, hence it may not be possible to design software for such systems according to the Unix Philosophy.

The Unix Philosophy has an idea of what the process of software development should look like, but large parts of the philosophy are quite independent from a concrete development process. However, one will soon recognize that some development processes work well with the ideas of the Unix Philosophy and support them, while others are at cross-purposes. Kent Beck's books about Extreme Programming are valuable supplemental resources on this topic.

The question of how to actually write code and how the code should look in detail, are beyond the scope of this paper. Kernighan and Pike's book "The Practice of Programming" [10] covers this topic. Its point of view corresponds to the one espoused in this paper.

2. Importance of software design in general

Software design consists of planning how the internal structure and external interfaces of software should look. It has nothing to do with visual appearance. If we were to compare a program to a car, then its color would not matter. Its design would be the car's size, its shape, the locations of doors, the passenger/space ratio, the available controls and instruments, and so forth.

Why should software be designed at all? It is accepted as general knowledge, that even a bad plan is better than no plan. Not designing software means programming without a plan. This will surely lead to horrible results, being horrible to use and horrible to maintain. These two aspects are the visible ones. Often invisible though, are the wasted possible gains. Good software design can make these gains available.

A software's design deals with qualitative properties. Good design leads to good quality, and quality is important. Any car may be able to drive from point A to point B, but it depends on the qualitative decisions made in the design of the vehicle, whether it is a good choice for passenger transport or not, whether it is a good choice for a rough mountain area, and whether the ride will be fun.

Requirements for a piece of software are twofold: functional and non-functional.

- Functional requirements directly define the software's functions. They are the reason why software gets written. Someone has a problem and needs a tool to solve it. Being able to solve the problem is the main functional goal. This is the driving force behind all programming effort. Functional requirements are easier to define and to verify.
- Non-functional requirements are called *quality* requirements, too. The quality of software shows through the properties that are not directly related to the software's basic functions. Tools of bad quality often do solve the problems they were written for, but introduce problems and difficulties for usage and development later on. Qualitative aspects are often overlooked at first sight, and are often difficult to define clearly and to verify.

Quality is hardly interesting when software gets built initially, but it has a high impact on usability and maintenance of the software later. A short-sighted person might see the process of developing software as one mainly concerned with building something up. But, experience shows that building software the first time is only a small portion of the overall work involved. Bug fixing, extending, rebuilding of parts – maintenance work – soon take a large part of the time spent on a software project. And of course, the time spent actually using the software. These processes are highly influenced by the software's quality. Thus, quality must not be neglected. However, the problem with quality is that you hardly “stumble over” bad quality during the first build, although this is the time when you should care about good quality most.

Software design has little to do with the basic function of software – this requirement will get satisfied anyway. Software design is more about quality aspects. Good design leads to good quality, bad design to bad quality. The primary functions of software will be affected modestly by bad quality, but good quality can provide a lot of additional benefits, even at places one never expected it.

The ISO/IEC 9126-1 standard, part 1 [9], defines the quality model as consisting of:

- *Functionality* (suitability, accuracy, interoperability, security)
- *Reliability* (maturity, fault tolerance, recoverability)
- *Usability* (understandability, learnability, operability, attractiveness)
- *Efficiency* (time behavior, resource utilization)

- *Maintainability* (analyzability, changeability, stability, testability)
- *Portability* (adaptability, installability, co-existence, replaceability)

Good design can improve these properties in software; poorly designed software likely suffers in these areas.

One further goal of software design is consistency. Consistency eases understanding, using, and working on things. Consistent internal structure and consistent external interfaces can be provided by good design.

Software should be well designed because good design avoids many problems during its lifetime. Also, because good design can offer much additional gain. Indeed, much effort should be spent on good design to make software more valuable. The Unix Philosophy provides a way to design software well. It offers guidelines to achieve good quality and high gain for the effort spent.

3. The Unix Philosophy

The origins of the Unix Philosophy have already been introduced. This chapter explains the philosophy, oriented on Gancarz [8], and shows concrete examples of its application.

3.1. Pipes

The following examples demonstrate how the Unix Philosophy is applied. Knowledge of using the Unix shell is assumed.

Counting the number of files in the current directory:

```
ls | wc -l
```

The `ls` command lists all files in the current directory, one per line, and `wc -l` counts the number of lines.

Counting the number of files that do not contain “foo” in their name:

```
ls | grep -v foo | wc -l
```

Here, the list of files is filtered by `grep` to remove all lines that contain “foo”. The rest equals the previous example.

Finding the five largest entries in the current directory:

```
du -s * | sort -nr | sed 5q
```

`du -s *` returns the recursively summed sizes of all files in the current directory – no matter if they are regular files or directories. `sort -nr` sorts the list numerically in reverse order (descending). Finally, `sed 5q` quits after it has printed the fifth line.

The presented command lines are examples of what Unix people would use to get the desired output. There are other ways to get the same output; it is the user’s decision which way to go.

The examples show that many tasks on a Unix system are accomplished by combining several small programs. The connection between the programs is denoted by the pipe operator ‘|’.

Pipes, and their extensive and easy use, are one of the great achievements of the Unix system. Pipes were possible in earlier operating systems, but never before have they been such a central part of the concept. In the early seventies when Doug McIlroy introduced pipes into

the Unix system, “it was this concept and notation for linking several programs together that transformed Unix from a basic file-sharing system to an entirely new way of computing.” [2]

Being able to specify pipelines in an easy way is, however, not enough by itself; it is only one half. The other is the design of the programs that are used in the pipeline. They need interfaces that allow them to be used in this way.

3.2. Interface design

Unix is, first of all, simple – everything is a file. Files are sequences of bytes, without any special structure. Programs should be filters, which read a stream of bytes from standard input (stdin) and write a stream of bytes to standard output (stdout). If the files *are* sequences of bytes, and the programs *are* filters on byte streams, then there is exactly one data interface. Hence it is possible to combine programs in any desired way.

Even a handful of small programs yields a large set of combinations, and thus a large set of different functions. This is leverage! If the programs are orthogonal to each other – the best case – then the set of different functions is greatest.

Programs can also have a separate control interface in addition to their data interface. The control interface is often called the “user interface”, because it is usually designed to be used by humans. The Unix Philosophy discourages the assumption that the user will be human. Interactive use of software is slow use of software, because the program waits for user input most of the time. Interactive software also requires the user to be in front of the computer, occupying his attention during usage.

Now, back to the idea of combining several small programs to perform a more specific function: If these single tools were all interactive, how would the user control them? It is not only a problem to control several programs at once if they run at the same time; it is also very inefficient to have to control each program when they are intended to act in concert. Hence, the Unix Philosophy discourages designing programs which demand interactive use. The behavior of programs should be defined at invocation. This is done by specifying arguments to the program call (command line switches). Gancarz discusses this topic as “avoid[ing] captive user interfaces” [8, page 88 ff.].

Non-interactive use is also an advantage for testing during development. Testing interactive programs is much more complicated than testing non-interactive counterparts.

3.3. The toolchest approach

A toolchest is a set of tools. Instead of one big tool for all tasks, there are many small tools, each for one task. Difficult tasks are solved by combining several small, simple tools.

The Unix toolchest *is* a set of small, (mostly) non-interactive programs that are filters on byte streams. They are, to a large extent, unrelated in their function. Hence, the Unix toolchest provides a large set of functions that can be accessed by combining the programs in the desired way.

The act of software development benefits from small toolchest programs, too. Writing small programs is generally easier and less error-prone than writing large programs. Hence, writing a large set of small programs is still easier and less error-prone than writing one large program with all the functionality included. If the small programs are combinable, then they offer even an even larger set of functions than the single monolithic program. Hence, one gets two advantages out of writing small, combinable programs: They are easier to write and they offer a greater set of functions through combination.

There are also two main drawbacks of the toolchest approach. First, one simple, standardized interface has to be sufficient. If one feels the need for more “logic” than a stream of bytes, then a different approach might be required. Also, a design where a stream of bytes is sufficient, might not be conceivable. By becoming more familiar with the “Unix style of thinking”, developers will more often and easier find simple designs where a stream of bytes is a sufficient interface.

The second drawback of the toolchest approach concerns the users. A toolchest is often more difficult to use because it is necessary to become familiar with each tool and be able to choose and use the right one in any given situation. Additionally, one needs to know how to combine the tools in a sensible way. The issue is similar to having a sharp knife – it is a powerful tool in the hand of a master, but of no value in the hand of an unskilled person. However, learning single, small tools of a toolchest is often easier than learning a complex tool. The user will already have a basic understanding of an as yet unknown tool if the tools of a toolchest have a common, consistent style. He will be able to transfer knowledge of one tool to another.

This second drawback can be removed to a large extent by adding wrappers around the basic tools. Novice users do not need to learn several tools if a professional wraps complete command lines into a higher-level script. Note that the wrapper script still calls the small tools; it is just like a skin around them. No complexity is added this way, but new programs can be created out of existing one with very little effort.

A wrapper script for finding the five largest entries in the current directory might look like this:

```
#!/bin/sh
du -s * | sort -nr | sed 5q
```

The script itself is just a text file that calls the commands that a professional user would type in directly. It is probably beneficial to make the program flexible in regard to the number of entries it prints:

```
#!/bin/sh
num=5
[ $# -eq 1 ] && num="$1"
du -sh * | sort -nr | sed "${num}q"
```

This script acts like the one before when called without an argument, but the user can also specify a numerical argument to define the number of lines to print. One can surely imagine even more flexible versions; however, they will still rely on the external programs which actually do the work.

3.4. A powerful shell

The Unix shell provides the ability to combine small programs into large ones. But a powerful shell is a great feature in other ways, too; for instance, by being scriptable. Control statements are built into the shell and the functions are the normal programs of the system. As the programs are already known, learning to program in the shell becomes easy. Using normal programs as functions in the shell programming language is only possible because they are small and combinable tools in a toolchest style.

The Unix shell encourages writing small scripts, by combining existing programs because it is so easy to do. This is a great step towards automation. It is wonderful if the effort to automate a task equals the effort to do the task a second time by hand. If this holds, then the user will be happy to automate everything he does more than once.

Small programs that do one job well, standardized interfaces between them, a mechanism to combine parts to larger parts, and an easy way to automate tasks will inevitably produce software leverage, achieving multiple times the benefit of the initial investment.

The shell also encourages rapid prototyping. Many well known programs started as quickly hacked shell scripts, and turned into “real” programs later written in C. Building a prototype first is a way to avoid the biggest problems in application development. Fred Brooks explains in “No Silver Bullet” [4]:

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is so difficult as establishing the detailed technical requirements, [...]. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

Writing a prototype is a great method for becoming familiar with the requirements and to run into real problems early [8, page 28 f.].

Prototyping is often seen as a first step in building software. This is, of course, good. However, the Unix Philosophy has an *additional* perspective on prototyping: After having built the prototype, one might notice that the prototype is already *good enough*. Hence, no reimplementing in a more sophisticated programming language might be of need, at least for the moment. Maybe later, it might be necessary to rewrite the software, but not now. By delaying further work, one keeps the flexibility to react on changing requirements. Software parts that are not written will not miss the requirements. Well known is Gordon Bell’s classic saying: “The cheapest, fastest, and most reliable components are those that aren’t there.”

3.5. Worse is better

The Unix Philosophy aims for the 90% solution; others call it the “Worse is better” approach. Experience from real life projects shows:

(1) It is almost impossible to define the requirements completely and correctly the first time. Hence one should not try to; one will fail anyway.

(2) Requirements change during time. Hence it is best to delay requirement-based design decisions as long as possible. Software should be small and flexible as long as possible in order to react to changing requirements. Shell scripts, for example, are more easily adjusted than C programs.

(3) Maintenance work is hard work. Hence, one should keep the amount of code as small as possible; it should only fulfill the *current* requirements. Software parts that will be written in the future do not need maintenance until that time.

See Brooks’ “The Mythical Man-Month” for reference [5, page 115 ff.].

Starting with a prototype in a scripting language has several advantages:

- As the initial effort is low, one will likely start right away.
- Real requirements can be identified quickly since working parts are available sooner.
- When software is usable and valuable, it gets used, and thus tested. This ensures that problems will be found in the early stages of development.
- The prototype might be enough for the moment; thus, further work can be delayed until a time when one knows about the requirements and problems more thoroughly.
- Implementing only the parts that are actually needed at the moment introduces less programming and maintenance work.
- If the situation changes such that the software is not needed anymore, then less effort was spent on the project than it would have been if a different approach had been taken.

3.6. Upgrowth and survival of software

So far, *writing* or *building* software has been discussed. Although “writing” and “building” are just verbs, they do imply a specific view on the work process they describe. A better verb would be to “grow”. Creating software in the sense of the Unix Philosophy is an incremental process. It starts with an initial prototype, which evolves as requirements change. A quickly hacked shell script might become a large, sophisticated, compiled program this way. Its lifetime begins with the initial prototype and ends when the software is not used anymore. While alive, it will be extended, rearranged, rebuilt. Growing software matches the view that “software is never finished. It is only released.” [8, page 26]

Software can be seen as being controlled by evolutionary processes. Successful software is software that is used by many for a long time. This implies that the software is necessary, useful, and better than the alternatives. Darwin describes “the survival of the fittest.” [7] In relation to software, the most successful software is the fittest; the one that survives. (This may be at the level of one creature, or at the level of one species.) The fitness of software is affected mainly by four properties: portability of code, portability of data, range of usability, and reusability of parts.

(1) “*Portability of code*” means using high-level programming languages, sticking to the standard [10, chapter 8], and avoiding optimizations that introduce dependencies on specific hardware. Hardware has a much shorter lifespan than software. By chaining software to specific hardware, its lifetime is limited to that of this hardware. In contrast, software should be easy to port – adaptation is the key to success.

(2) “*Portability of data*” is best achieved by avoiding binary representations to store data, since binary representations differ from machine to machine. Textual representation is favored. Historically, ASCII was the character set of choice; for the future, UTF-8 might be the better way forward. Important is that it is a plain text representation in a very common character set encoding. Apart from being able to transfer data between machines, readable data has the great advantage that humans are able to directly read and edit it with text editors and other tools from the Unix toolchest [8, page 56 ff.].

(3) A large “*range of usability*” ensures good adaptation, and thus good survival. It is a special distinction when software becomes used in fields of endeavor, the original authors never imagined. Software that solves problems in a general way will likely be used for many kinds of similar problems. Being too specific limits the range of usability. Requirements change through time, thus use cases change or even vanish. As a good example of this point, Allman identifies flexibility to be one major reason for sendmail’s success [6]:

Second, I limited myself to the routing function [...]. This was a departure from the dominant thought of the time, [...].

Third, the sendmail configuration file was flexible enough to adapt to a rapidly changing world [...].

Successful software adapts itself to the changing world.

(4) “*Reusability of parts*” goes one step further. Software may become obsolete and completely lose its field of action, but the constituent parts of the software may be general and independent enough to survive this death. If software is built by combining small independent programs, then these parts are readily available for reuse. Who cares that the large program is a failure, if parts of it become successful instead?

3.7. Summary

This chapter explained ideas central to the Unix Philosophy. For each of the ideas, the advantages they introduce were explained. The Unix Philosophy is a set of guidelines that help in the design of more valuable software. From the viewpoint of a software developer or software designer, the Unix Philosophy provides answers to many software design problems.

The various ideas comprising the Unix Philosophy are very interweaved and can hardly be applied independently. The most important messages are: “*Keep it simple!*”, “*Do one thing well!*”, and “*Use software leverage!*”

4. Case study: MH

The previous chapter introduced and explained the Unix Philosophy from a general point of view. The driving force was that of the guidelines; references to existing software were given only sparsely. In this and the next chapter, concrete software will be the driving force in the discussion.

This first case study is about the mail user agents (MUA) MH (“mail handler”) and its descendant *nmh* (“new mail handler”) [13]. MUAs provide functions to read, compose, and organize mail, but (ideally) not to transfer it. In this document, the name MH will be used to include *nmh*. A distinction will only be made if differences between MH and *nmh* are described.

4.1. Historical background

Electronic mail was available in Unix from a very early stage. The first MUA on Unix was *mail*, which was already present in the First Edition [17, page 41 f.]. It was a small program that either printed the user’s mailbox file or appended text to someone else’s mailbox file, depending on the command line arguments [18]. It was a program that did one job well. This job was emailing, which was very simple then.

Later, emailing became more powerful, and thus more complex. The simple *mail*, which knew nothing of subjects, independent handling of single messages, and long-term email storage, was not powerful enough anymore. In 1978 at Berkeley, Kurt Shoens wrote *Mail* (with a capital ‘M’) to provide additional functions for emailing. *Mail* was still one program, but was large and did several jobs. Its user interface was modeled after *ed*. *Ed* is designed for humans, but is still scriptable. *mailx* is the adaptation of Berkeley Mail for System V [16]. Elm, pine, mutt, and a slew of graphical MUAs followed *Mail*’s direction: large, monolithic programs which included all emailing functions.

A different way was taken by the people of RAND Corporation. Initially, they also had used a monolithic mail system called MS (for “mail system”). But in 1977, Stockton Gaines and Norman Shapiro came up with a proposal for a new email system concept – one that honored the Unix Philosophy. The concept was implemented by Bruce Borden in 1978 and 1979. This was the birth of MH – the “mail handler”.

Since then, RAND, the University of California at Irvine and at Berkeley, and several others have contributed to the software. However, it’s core concepts remained the same. In the late 90s, when development of MH slowed down, Richard Coleman started with *nmh*, the new mail handler. His goal was to improve MH especially in regard to the requirements of modern emailing. Today, *nmh* is developed by various people on the Internet [14, 22].

4.2. Contrasts to monolithic mail systems

All MUAs are monolithic, except MH. Although some very little known toolchest MUAs might also exist, this statement reflects the situation pretty well.

Monolithic MUAs gather all their functions in one program. In contrast, MH is a toolchest of many small tools – one for each job. Following is a list of important programs of MH’s toolchest and their function. It gives an indication of what the toolchest looks like.

- `inc`: incorporate new mail (this is how mail enters the system)
- `scan`: list messages in folder
- `show`: show message
- `next/prev`: show next/previous message
- `folder`: change current folder
- `refile`: refile message into different folder
- `rmm`: remove message
- `comp`: compose new message
- `repl`: reply to message
- `forw`: forward message
- `send`: send prepared message (this is how mail leaves the system)

MH has no special user interface like monolithic MUAs have. The user does not leave the shell to run MH; instead he uses the various MH programs within the shell. Using a monolithic program with a captive user interface means “entering” the program, using it, and “exiting” the program. Using toolchests like MH means running programs, alone or in combination with others, also from other toolchests, without leaving the shell.

4.3. Data storage

MH’s mail storage consists of a hierarchy under the user’s MH directory (usually `$HOME/Mail`), where mail folders are directories and mail messages are text files within them. Each mail folder contains a file `.mh_sequences` which lists the public message sequences of that folder, for instance, the *unseen* sequence for new messages. Mail messages are text files located in a mail folder. The files contain the messages as they were received, and they are named by ascending numbers in each folder.

This mailbox format is called “MH” after the MUA. Alternatives are *mbox* and *maildir*. In the *mbox* format, all messages are stored within one file. This was a good solution in the early days, when messages were only a few lines of text deleted within a short period of time. Today, with single messages often including several megabytes of attachments, this is a bad solution. Another disadvantage of the *mbox* format is that it is more difficult to write tools that work on mail messages, because it is always necessary to first find and extract the relevant message in the *mbox* file. With the MH mailbox format, each message is a separate file. Also, the problem of concurrent access to one mailbox is reduced to the problem of concurrent access to one message. The *maildir* format is generally similar to the MH format, but modified towards guaranteed reliability. This involves some complexity, unfortunately.

Working with MH’s toolchest on mailboxes is much like working with Unix’ toolchest on directory trees: `scan` is like `ls`, `show` is like `cat`, `folder` is like `cd` and `pwd`, `refile` is like `mv`, and `rmm` is like `rm`.

MH extends the context of processes in Unix by two more items for its tools:

- The current mail folder, which is similar to the current working directory. For mail folders, `folder` provides the corresponding functionality of `cd` and `pwd` for directories.
- Sequences, which are named sets of messages in a mail folder. The current message, relative to a mail folder, is a special sequence. It enables commands like `next` and `prev`.

In contrast to the general process context in Unix, which is maintained by the kernel, MH's context must be maintained by the tools themselves. Usually there is one context per user, which resides in his `context` file in the MH directory, but a user can have several contexts, too. Public sequences are an exception, as they belong to a mail folder, and reside in the `.mh_sequences` file there [12].

4.4. Discussion of the design

This section discusses MH in regard to the tenets of the Unix Philosophy that Gancarz identified.

Small is beautiful and **do one thing well** are two design goals that are directly visible in MH. Gancarz actually uses MH in his book as example under the headline “Making UNIX Do One Thing Well” [8, page 125 ff.]:

[MH] consists of a series of programs which when combined give the user an enormous ability to manipulate electronic mail messages. A complex application, it shows that not only is it possible to build large applications from smaller components, but also that such designs are actually preferable.

The various programs of MH were relatively easy to write, because each one was small, limited to one function, and had clear boundaries. For the same reasons, they are also easy to maintain. Further more, the system can easily get extended: One only needs to place a new program into the toolchest. This was done when MIME support was added (e.g. `mhbuild`). Also, different programs can exist to do basically the same job in different ways (e.g. in `nmh`: `show` and `mhshow`).

If someone needs a mail system with some additional functionality that is not available anywhere yet, it is beneficial to expand a toolchest system like MH. There he can add new functionality by simply adding additional programs to the toolchest; he does not risk to break existing functionality by doing so.

Store data in flat text files; this principle was followed by MH. This is not surprising, because email messages are already plain text. MH stores the messages as it receives them, thus any other tool that works on RFC 2822 compliant mail messages can operate on the messages in an MH mailbox. All other files MH uses are plain text as well. It is therefore possible and encouraged to use the text processing tools of Unix' toolchest to extend MH's toolchest.

Avoid captive user interfaces. MH is perfectly suited for non-interactive use. It offers all functions directly, without captive user interfaces. If users want a graphical user interface, they can have it with `xmh`, `exmh`, or with the Emacs interface `mh-e`. These are frontends for the MH toolchest. This means all email-related work is still done by MH tools, but the frontend calls the appropriate commands when the user clicks on buttons or pushes a key.

Providing additional user interfaces in form of frontends is a good approach, because it does not limit the power of the backend itself. The frontend will only be able to make a subset of the backend's power and flexibility available to the user, but if it is a separate program, then the missing parts can still be accessed at the backend directly. If it is integrated, then this

will be much more difficult. An additional advantage is the ability to have different frontends to the same backend.

Choose portability over efficiency and use shell scripts to increase leverage and portability. These two tenets are indirectly, but nicely, demonstrated by Bolsky and Korn in their book about the Korn Shell [3]. Chapter 18 of the book shows a basic implementation of a subset of MH in ksh scripts. This is just a demonstration, but a brilliant one. It shows how quickly one can implement such a prototype with shell scripts, and how readable they are. The implementation in scripting language may not be very fast, but it can be fast enough, and this is all that matters. By having the code in an interpreted language, like the shell, portability becomes a minor issue if we assume the interpreter to be widespread.

This demonstration also shows how easy it is to create single programs of toolchest software. Eight tools (two of them having multiple names) and 16 functions with supporting code are presented to the reader. The tools comprise less than 40 lines of ksh each, in total about 200 lines. The functions comprise less than 80 lines of ksh each, in total about 450 lines. Such small software is easy to write, easy to understand, and thus easy to maintain. A toolchest improves one's ability to only write some parts of a program while still creating a working result. Expanding the toolchest, even without global changes, will likely be possible.

Use software leverage to your advantage and the lesser tenet **allow the user to tailor the environment** are ideally followed in the design of MH. Tailoring the environment is heavily encouraged by the ability to directly define default options to programs. It is even possible to define different default options depending on the name under which a program is called. Software leverage is heavily encouraged by the ease of creating shell scripts that run a specific command line, built of several MH programs. There are few pieces of software that encourages users to tailor their environment and to leverage the use of the software like MH.

Just to cite one example: One might prefer a different listing format for the `scan` program. It is possible to take one of the distributed format files or to write one yourself. To use the format as default for `scan`, a single line, reading

```
scan: -form FORMATFILE
```

must be added to `.mh_profile`. If one wants this alternative format available as an additional command, instead of changing the default, he just needs to create a link to `scan`, for instance titled `scan2`. The line in `.mh_profile` would then start with `scan2`, as the option should only be in effect for a program that is invoked as `scan2`.

Make every program a filter is hard to find implemented in MH. The reason is that most of MH's tools provide basic file system operations for mailboxes. It is for the same reason because that `ls`, `cp`, `mv`, and `rm` aren't filters neither. MH does not provide many filters itself, but it provides a basis upon which to write filters. An example would be a mail text highlighter, a program that makes use of a color terminal to display header lines, quotations, and signatures in distinct colors. The author's version of such a program is an awk script with 25 lines.

Build a prototype as soon as possible was again well followed by MH. This tenet, of course, focuses on early development, which is a long time ago for MH. But without following this guideline at the very beginning, Bruce Borden may have not convinced the management of RAND to ever create MH. In Bruce' own words [22, page 132]:

[...] but [Stockton Gaines and Norm Shapiro] were not able to convince anyone that such a system would be fast enough to be usable. I proposed a very short project to prove the basic concepts, and my management agreed. Looking back, I realize that I had been very lucky with my first design. Without nearly enough design work, I built a working

environment and some header files with key structures and wrote the first few MH commands: `inc`, `show/next/prev`, and `comp`. [...] With these three, I was able to convince people that the structure was viable. This took about three weeks.

4.5. Problems

MH is not without its problems. There are two main problems: one is technical, the other pertains to human behavior.

MH is old and email today is quite different than it was in the time when MH was designed. MH adapted to the changes fairly well, but it has its limitations. MIME support and support for different character encodings is available, but only on a moderate level. This comes from limited development resources. A larger and more active developer base could quickly remedy this. But MH is also limited by design, which is the larger problem. IMAP, for example, conflicts with MH's design to a large extent. These design conflicts are not easily solvable and may require a redesign. IMAP may be too incompatible with the classic mail model, which MH covers, so MH may never support it well. (Using IMAP and a filesystem abstraction layer to only map a remote directory into the local filesystem, is a different topic. IMAP support is seen as being able to access the special mail features of the protocol.)

The other kind of problem relates to human habits. In this world, where almost all MUAs are monolithic, it is very difficult to convince people to use a toolchest-style MUA like MH. These habits are so strong, that even people who understand the concept and advantages of MH are reluctant to switch, simply because MH is different. Unfortunately, the frontends to MH, which could provide familiar look and feel, are quite outdated and thus not very appealing in comparison to the modern interfaces of many monolithic MUAs. One notable exception is *mh-e* which provides an Emacs interface to MH. *Mh-e* looks much like *mutt* or *pine*, but it has buttons, menus, and graphical display capabilities.

4.6. Summary

MH is an MUA that follows the Unix Philosophy in its design. It consists of a toolchest of small tools, each of which does one job well. The toolchest approach offers great flexibility to the user. It is possible to utilize the complete power of the Unix shell with MH. This makes MH a very powerful mail system, and extending and customizing MH is easy and encouraged.

Apart from the user's perspective, MH is development-friendly. Its overall design follows clear rules. The single tools do only one job; thus they are easy to understand, write, and maintain. They are all independent and do not interfere with the others. Automated testing of their function is a straightforward task.

It is sad, that MH's dissimilarity to other MUAs is its largest problem, as this dissimilarity is also its largest advantage. Unfortunately, most people's habits are stronger than the attraction of the clear design and the power MH offers.

5. Case study: uzbl

The last chapter focused on the MUA MH, which is an old and established piece of software. This chapter covers uzbl, a fresh new project. Uzbl is a web browser that adheres to the Unix Philosophy. Its name comes from the *Lolspeak* word for "usable"; both are pronounced in the same way.

5.1. Historical background

Uzbl was started by Dieter Plaetinck in April 2009. The idea was born in a thread on the Arch Linux forums [1]. After some discussion about the failures of well-known web browsers, Plaetinck (alias Dieter@be) came up with a rough proposal of how a better web browser could look. In response to another member who asked if Plaetinck would write this program because it sounded fantastic, Plaetinck replied: “Maybe, if I find the time ;-”).

Fortunately, he found the time. One day later, the first prototype was out. One week later, uzbl had its own website [20]. One month after the initial code was presented, a mailing list was set up to coordinate and discuss further development, and a wiki was added to store documentation and scripts that cropped up on the mailing list and elsewhere.

In the first year of uzbl’s existence, it was heavily developed on various branches. Plaetinck’s task became more and more to only merge the best code from the different branches into his main branch, and to apply patches [21]. About once a month, Plaetinck released a new version. In September 2009, he presented several forks of uzbl [20, news archive]. Uzbl actually opened the field for a whole family of web browsers with a similar design.

In July 2009, *Linux Weekly News* published an interview with Plaetinck about uzbl [21]. In September 2009, the uzbl web browser was on *Slashdot* [19].

5.2. Contrasts to other web browsers

Like most MUAs are monolithic, but MH is a toolchest, most web browsers are monolithic, but uzbl is a frontend to a toolchest.

Today, uzbl is divided into uzbl-core and uzbl-browser. Uzbl-core is, as its name indicates, the core of uzbl. It handles commands and events to interface with other programs, and displays webpages by using *webkit* as its rendering engine. Uzbl-browser combines uzbl-core with a selection of handler scripts, a status bar, an event manager, yanking, pasting, page searching, zooming, and much more functionality, to form a “complete” web browser. In the following text, the term “uzbl” usually refers to uzbl-browser, so uzbl-core is included.

Unlike most other web browsers, uzbl is mainly the mediator between various tools that cover single jobs. Uzbl listens for commands on a named pipe (fifo), a Unix socket, and on stdin, and it writes events to a Unix socket and to stdout. Loading a webpage in a running uzbl instance requires only:

```
echo 'uri http://example.org' >/path/to/uzbl-fifo
```

The rendering of the webpage is done by libwebkit, around which uzbl-core is built.

Downloads, browsing history, bookmarks, and the like are not provided by the core itself like they are in other web browsers. Uzbl-browser also only provides “handler scripts” which wrap external applications to provide the actual functionality. For instance, *wget* is used to download files and uzbl-browser includes a script that calls wget with appropriate options in a prepared environment.

Modern web browsers are proud to have addons, plugins, modules, and so forth. This is their effort to achieve similar goals. But instead of using existing external programs, modern web browsers include these functions.

5.3. Discussion of the design

This section discusses uzbl in regard to the Unix Philosophy, as identified by Gancarz.

Make each program do one thing well. Uzbl tries to be a web browser and nothing else. The common definition of a web browser is highly influenced by existing implementations of web browsers. But a web browser should be a program to browse the web, and nothing more. This is the one thing it should do.

Web browsers should not, for instance, manage downloads; this is the job of download managers. A download manager is primary concerned with downloading files. Modern web browsers provide download management only as a secondary feature. How could they do this job better than programs that exist only for this very job? And why would anyone want less than the best download manager available?

A web browser's job is to let the user browse the web. This means, navigating through websites by following links. Rendering the HTML sources is a different job, too. In uzbl's case, this is covered by the webkit rendering engine. Handling audio and video content, PostScript, PDF, and other such files are also not the job of a web browser. Such content should be handled by external programs that were written to handle such data. Uzbl strives to do it this way.

Remember Doug McIlroy's words: *"Write programs that do one thing and do it well. Write programs to work together."*

The lesser tenet **allow the user to tailor the environment** applies here as well. Previously, the question, "Why would anyone want anything less than the best program for the job?" was put forward. But as personal preferences matter, it might be more important to ask: "Why would anyone want something other than his preferred program for the job?"

Users typically want one program for a specific job. Hence, whenever one wishes to download something, the same download manager should be used. More advanced users might want to use one download manager in a certain situation and another in a different situation; they should be able to configure it this way. With uzbl, any download manager can be used. To switch to a different one, a single line in a small handler script needs to be changed. Alternatively, it would be possible to query which download manager to use by reading a global file or an environment variable in the handler script. Of course, uzbl can use a different handler script as well. This simply requires a one line change in uzbl's configuration file.

Uzbl neither has its own download manager nor depends on a specific one; hence, uzbl's browsing abilities will not be crippled by having a bad download manager. Uzbl's download capabilities will be as good as the best download manager available on the system. Of course, this applies to all of the other supplementary tools, too.

Use software leverage to your advantage. Uzbl is designed to be extended by external tools. These external tools are usually wrapped by small handler shell scripts. Shell scripts form the basis for the glue which holds the various parts together.

The history mechanism of uzbl shall be presented as an example. Uzbl is configured to spawn a script to append an entry to the history whenever the event of a fully loaded page occurs. The script to append the entry to the history is not much more than:

```
#!/bin/sh
file=/path/to/uzbl-history
echo `date +%Y-%m-%d %H:%M:%S` "$6 $7" >> $file
```

\$6 and \$7 expand to the URL and the page title, respectively.

For loading an entry, a key is bound to spawn a load-from-history script. The script reverses the history to have newer entries first, displays *dmenu* to let the user select an item, and then writes the selected URL into uzbl's command input pipe. With error checking and corner case handling removed, the script looks like this:

```
#!/bin/sh
file=/path/to/uzbl-history
goto=`tac $file | dmenu | cut -d' ' -f 3`
echo "uri $goto" > $4
```

\$4 expands to the path of the command input pipe of the current uzbl instance.

Avoid captive user interfaces. One could say that uzbl, to a large extent, actually *is* a captive user interface. But the difference from other web browsers is that uzbl is only the captive user interface frontend (and the core of the backend). Many parts of the backend are independent of uzbl. For some external programs, handler scripts are distributed with uzbl; but arbitrary additional functionality can always be added if desired.

The frontend is captive – that is true. This is okay for the task of browsing the web, as this task is only relevant to humans. Automated programs would *crawl* the web, that means, read the source directly, including all semantics. The graphical representation is just for humans to understand the semantics more intuitively.

Make every program a filter. Graphical web browsers are almost dead ends in the chain of information flow. Thus it is difficult to see what graphical web browsers should filter. Graphical web browsers exist almost exclusively to be interactively used by humans. The only case in which one might want to automate the rendering function is to generate images of rendered webpages.

Small is beautiful is not easy to apply to a web browser because modern web technology is very complex; hence, the rendering task is very complex. Unfortunately, modern web browsers “have” to consist of many thousand lines of code, Using the toolchest approach and wrappers can help to split the browser into several small parts, though.

As of March 2010, uzbl-core consists of about 3,500 lines of C code. The distribution includes another 3,500 lines of Shell and Python code, which are the handler scripts and plugins like one to provide a modal interface. Further more, uzbl makes use of external tools like *wget* and *socat*. Up to this point, uzbl looks pretty neat and small. The ugly part of uzbl is the rendering engine, webkit. Webkit consists of roughly 400,000 (!) lines of code. Unfortunately, small rendering engines are not feasible anymore due to the nature of the modern web.

Build a prototype as soon as possible. Plaetinck made his code public right from the beginning. Discussion and development was, and still is, open to everyone interested, and development versions of uzbl can be obtained very easily from the code repository. Within the first year of uzbl's existence, a new version was released more often than once a month. Different forks and branches arose introducing new features which were then considered for merging into the main branch. The experiences with using prototypes influenced further development. Actually, all development was community driven. Plaetinck says, three months after uzbl's birth: “Right now I hardly code anything myself for Uzbl. I just merge in other people's code, ponder a lot, and lead the discussions.” [21]

5.4. Problems

Similar to MH, uzbl suffers from being different. It is sad, but people use what they know. Fortunately, uzbl's user interface can be made to look and feel very similar to the one of the well known web browsers, hiding the internal differences. But uzbl has to provide this similar look and feel to be accepted as a "normal" browser by "normal" users.

The more important problem here is the modern web. The modern web is simply broken. It has state in a state-less protocol, misuses technologies, and is helplessly overloaded. This results in rendering engines that "must" consist of hundreds of thousands of lines of code. They also must combine and integrate many different technologies to make our modern web accessible. This results, however, in a failing attempt to provide good usability. Website-to-image converters are almost impossible to run without human interaction because of state in sessions, impossible deep-linking, and "unautomatable" technologies.

The web was misused in order to attempt to fulfill all kinds of wishes. Now web browsers, and ultimately users, suffer from it.

5.5. Summary

"Uzbl is a browser that adheres to the Unix Philosophy", is how uzbl is seen by its authors. Indeed, uzbl follows the Unix Philosophy in many ways. It consists of independent parts that work together, while its core is mainly a mediator which glues the parts together.

Software leverage is put to excellent use. External tools are used, independent tasks are separated out to independent parts and glued together with small handler scripts.

Since uzbl roughly consists of a set of tools and a bit of glue, anyone can put the parts together and expand it in any desired way. Flexibility and customization are properties that make it valuable for advanced users, but may keep novice users from understanding and using it.

But uzbl's main problem is the modern web, which makes it very difficult to design a sane web browser. Despite this bad situation, uzbl does a fairly good job.

6. Final thoughts

This paper explained why good design is important. It introduced the Unix Philosophy as a set of guidelines that encourage good design in order to create good quality software. Then, real world software that was designed with the Unix Philosophy in mind was discussed.

Throughout this paper, the aim was do explain *why* something should be done the Unix way. Reasons were given to substantiate the claim that the Unix Philosophy is a preferable way of designing software.

The Unix Philosophy is close to the software developer's point of view. Its main goal is taming the beast known as "software complexity". Hence it strives first and foremost for simplicity of software. It might appear that usability for humans is a minor goal, but actually, the Unix Philosophy sees usability as a result of sound design. Sound design does not need to be ultimately intuitive, but it will provide a consistent way to access the enormous power of software leverage.

Being able to solve some specific concrete problem becomes less and less important as there is software available for nearly every possible task today. But the quality of software matters. It is important that we have *good* software.

But why the Unix Philosophy?

The largest problem of software development is the complexity involved. It is the only part of the job that computers cannot take over. The Unix Philosophy fights complexity, as it is the main enemy.

On the other hand, the most unique advantage of software is its ability to leverage. Current software still fails to make the best possible use of this ability. The Unix Philosophy concentrates on exploiting this great opportunity.

References

1. Arch Linux Forums, *Thread "Arch Philosophy/Structure Applied to a Browser"*, Spring 2009. Online: <http://bbs.archlinux.org/viewtopic.php?id=67463>
2. Jason Aughenbaugh, Jonathan Jessup, and Nicholas Spicher, "Building Unix," in *Unix: An Oral History*. Online: <http://www.princeton.edu/~hos/frs122/unixhist/finalhis.htm>
3. Morris I. Bolsky and David G. Korn, *The KornShell: command and programming language*, p. 254–290, Prentice Hall, 1989. ISBN: 0-13-516972-0
4. Frederick P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," in *Information Processing 1986, the Proceedings of the IFIP Tenth World Computing Conference*, p. 1069–1076, Elsevier Science B.V., Amsterdam, The Netherlands, 1986.
5. Frederick P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition, Addison Wesley Longman, Inc., 1995. ISBN: 0-201-83595-9
6. Bryan Costales and Eric Allman, *sendmail*, p. xix, O'Reilly, 2003. ISBN: 1-56592-839-3
7. Charles Darwin, *On the Origin of Species*, John Murray, London, 1859. Available online: [http://en.wikisource.org/wiki/On_the_Origin_of_Species_\(1859\)](http://en.wikisource.org/wiki/On_the_Origin_of_Species_(1859))
8. Mike Gancarz, *The UNIX Philosophy*, Digital Press, 1995. ISBN: 1-55558-123-4
9. *ISO Standard 9126: Software Engineering – Product Quality, part 1*, International Organization for Standardization, Geneva, 2001.
10. Brian W. Kernighan and Rob Pike, *The Practice of Programming*, Addison-Wesley, 1999. ISBN: 0-201-61586-X
11. Michael S. Mahoney, *The UNIX Oral History Project*, Bell Laboratories. Online: <http://www.princeton.edu/~hos/Mahoney/expotape.htm>
12. MH/nmh workers, *MH/nmh Documentation*, pp. mh-profile(5), mh-sequence(5). Distributed with nmh-1.3. Online in possibly different versions: <http://linux.die.net/man/5/mh-profile>
<http://linux.die.net/man/5/mh-sequence>
13. *Website of nmh*. Online: <http://nmh.nongnu.org>
14. Jerry Peek, *MH & xmh: Email for Users & Programmers*, p. Appendix B, O'Reilly, 1995. Also available online: <http://rand-mh.sourceforge.net/book/>
15. Eric S. Raymond, *The Art of UNIX Programming*, Addison-Wesley, 2003. Also available online: <http://www.faqs.org/docs/artu/>
16. Gunnar Ritter, *mail, Mail, mailx, nail—history notes*, 2007-01-28. Online: http://heirloom.sourceforge.net/mailx_history.html
17. Peter H. Salus, *A Quarter Century of UNIX*, Addison-Wesley, 1994. ISBN: 0-201-54777-5
18. Ken Thompson and Dennis M. Ritchie, *Unix Programmer's Manual*, First Edition, p. mail(1), 1971-11-03. Online: <http://cm.bell-labs.com/cm/cs/who/dmr/pdfs/man12.pdf>
19. DigDuality (posted by timothy), *Meet Uzbl – a Web Browser With the Unix Philosophy*, Slashdot, 2009-09-05. Online: <http://linux.slashdot.org/story/09/09/05/2142235>
20. *Website of uzbl*. Online: <http://uzbl.org>
21. Koen Vervloesem, *Uzbl: a browser following the UNIX philosophy*, LWN.net, 2009-07-15. Online: <http://lwn.net/Articles/341245/>
22. Willis H. Ware, *RAND and the Information Evolution: A History in Essays and Vignettes*, p. 128–137, The RAND Corporation, 2008. ISBN: 978-0-8330-4513-3. Also available online: http://www.rand.org/pubs/corporate_pubs/CP537/
23. Wikipedia, The Free Encyclopedia, *Unix philosophy*, Version of 2010-03-21 17:20 UTC. Online: http://en.wikipedia.org/w/index.php?title=Unix_philosophy&oldid=351189719

Table of Contents

1. Introduction	1
2. Importance of software design in general	2
3. The Unix Philosophy	3
3.1. Pipes	3
3.2. Interface design	4
3.3. The toolchest approach	4
3.4. A powerful shell	5
3.5. Worse is better	6
3.6. Upgrowth and survival of software	7
3.7. Summary	8
4. Case study: MH	8
4.1. Historical background	8
4.2. Contrasts to monolithic mail systems	9
4.3. Data storage	9
4.4. Discussion of the design	10
4.5. Problems	12
4.6. Summary	12
5. Case study: uzbl	12
5.1. Historical background	13
5.2. Contrasts to other web browsers	13
5.3. Discussion of the design	14
5.4. Problems	16
5.5. Summary	16
6. Final thoughts	16
References	18