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 no limited set of fixed rules, but a loose set of guidelines which tell how to write software that suites well into Unix. Actually, the Unix Philosophy describes what is common to 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].

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, base on facilities of Unix. Other operating systems may not offer such facilities, hence it may not be possible to design software in the way of the Unix Philosophy for them.

The Unix Philosophy has an idea of how 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 looks like in detail, are out of focus here. “The Practice of Programming” by Kernighan and Pike [10], is a good book that covers this topic. Its point of view matches to the one of this paper.
2. Importance of software design in general

Software design is the planning of how the internal structure and external interfaces of a software should look like. It has nothing to do with visual appearance. If we take a program as a car, then its color is of no 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 get designed at all? It is general knowledge, that even a bad plan is better than no plan. Not designing software means programming without plan. This will pretty sure lead to horrible results. Software that is 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 quality properties. Good design leads to good quality, and quality is important. Any car may be able to drive from A to B, but it depends on the car’s properties whether it is a good choice for passenger transport or not. It depends on its properties if it is a good choice for a rough mountain area. And it depends on its properties if the ride will be fun.

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

- Functional requirements define directly 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. It 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 a software are 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. Quality aspects are often overlooked at first sight, and are often difficult to define clearly and to verify.

Quality is hardly interesting when the software gets built initially, but it has a high impact on usability and maintenance of the software, later. A short-sighted might see in developing a software, mainly building something up. But experience shows, that building the software the first time is only a small amount of the overall work. Bug fixing, extending, rebuilding of parts – maintenance work – does soon take over the major part of the time spent on a software. 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 a software – this requirement will get satisfied anyway. Software design is more about quality aspects of the software. Good design leads to good quality, bad design to bad quality. The primary functions of the software will be affected modestly by bad quality, but good quality can provide a lot of additional gain, even at places where one never expected it.

The ISO/IEC 9126-1 standard, part 1 [9], defines the quality model as consisting out 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 of a software, bad designed software likely suffers in these points.

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

Software should be well designed because good design avoids many problems during a software's lifetime. And software should be well designed because good design can offer much additional gain. Indeed, much effort should be spent into good design to make software more valuable. The Unix Philosophy shows a way of how 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 were already introduced. This chapter explains the philosophy, oriented on Gancarz, and shows concrete examples of its application.

3.1. Pipes

Following are some examples to demonstrate how applied Unix Philosophy feels like. 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 also other ways to get the same output. It’s a 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 single 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 between programs have been possible in earlier operating systems, but it has never been a so central part of the concept. When, in the early seventies, 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 such a 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, besides their data interface. The control interface is often called “user interface”, because it is usually designed to be used by humans. The Unix Philosophy discourages to assume the user to be human. Interactive use of software is slow use of software, because the program waits for user input most of the time. Interactive software requires the user to be in front of the computer. Interactive software occupy the user’s attention while they are running.

Now to come back to the idea of combining several small programs, to have a more specific function. If these single tools would all be 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 also very inefficient to have to control each of the single programs that are intended to act as one large program. Hence, the Unix Philosophy discourages programs to 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 captive user interfaces” [8, page 88 ff.].

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

3.3. The toolchest approach

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

There are also advantages for developing small toolchest programs. It is easier and less error-prone to write small programs. It is also easier and less error-prone to write a large set of small programs, than to write one large program with all the functionality included. If the small programs are combinable, then they offer even a larger set of functions than the single large 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.
But 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 of need. But it is also possible, that he just can not imagine a design where a stream of bytes is sufficient. 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 a toolchest affects the users. A toolchest is often more difficult to use. It is necessary to become familiar with each of the tools, to be able to use the right one in a given situation. Additionally, one needs to combine the tools in a sensible way himself. This is like a sharp knife – it is a powerful tool in the hand of a master, but of no good value in the hand of an unskilled. However, learning single, small tools of a toolchest is easier than learning a complex tool. And the user will already have a basic understanding of a yet unknown tool, if the tools of a toolchest have a common, consistent style. He will be able to transfer knowledge over from one tool to another.

Moreover, the 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 get created out of existing one with very low effort.

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

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

The script itself is just a text file that calls the command line, which a professional user would type in directly. It is probably worth to make the program flexible on 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 one 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 do the actual work.

### 3.4. A powerful shell

The Unix shell provides the possibility 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. The functions, however, are the normal programs of the system. Thus, 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 to write 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, this will inevitably produce software leverage. Getting multiple times the benefit of an investment is a great offer.

The shell also encourages rapid prototyping. Many well known programs started as quickly hacked shell scripts, and turned into "real" programs, written in C, later. 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 a 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 reimplementatio, 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.

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 never possible 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. The software should be small and flexible as long as possible to react on changing requirements. Shell scripts, for example, are more easily adjusted as C programs.

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

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.
- As working parts are available soon, the real requirements can get identified soon.
- When a software is usable and valuable, it gets used, and thus tested. Hence problems will be found at early stages of the development.
- The prototype might be enough for the moment, thus further work can get delayed to a time when one knows better about the requirements and problems, than now.
- Implementing now only the parts that are actually needed at the moment, introduces fewer programming and maintenance work.
- If the global situation changes so that the software is not needed anymore, then less effort was spent into the project, than it would have be when a different approach had been used.
3.6. Upgrowth and survival of software

So far it was talked about writing or building software. Although these are just verbs, they do imply a specific view on the work process they describe. The better verb, however, is to grow. Creating software in the sense of the Unix Philosophy is an incremental process. It starts with a first 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 being alive it will get 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 needed, useful, and better than alternatives. Darwin talks about: “The survival of the fittest.” [7] Transferred to software: The most successful software, is the fittest, is 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 lower lifetime than software. By chaining software to a specific hardware, the software’s lifetime gets shortened 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, because binary representations differ from machine to machine. Textual representation is favored. Historically, ASCII was the charset of choice. For the future, UTF-8 might be the better choice. Important is that it is a plain text representation in a very common charset 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 if a software becomes used in fields of action, the original authors did never imagine. 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 in 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) Reuse of parts is even one step further. A software may completely lose its field of action, but parts of which the software is build may be general and independent enough to survive this death. If software is build by combining small independent programs, then these parts are readily available for reuse. Who cares if the large program is a failure, but parts of it become successful instead?
3.7. Summary

This chapter explained central ideas of the Unix Philosophy. For each of the ideas, the advantages they introduce were explained. The Unix Philosophy are guidelines that help to write 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 of the Unix Philosophy are very interweaved and can hardly be applied independently. However, the probably 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 were 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 descendent 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 very early. 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-time email storage, was not powerful enough anymore. In 1978 at Berkeley, Kurt Shoens wrote Mail (with capital ‘M’) to provide additional functions for emailing. Mail was still one program, but now it was large and did several jobs. Its user interface is modeled after the one of ed. It is designed for humans, but is still scriptable. mailx is the adaptation of Berkeley Mail into System V [16]. Elm, pine, mutt, and a whole bunch of graphical MUAs followed Mail’s direction. They are large, monolithic programs which include all emailing functions.

A different way was taken by the people of RAND Corporation. In the beginning, 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 of 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 of 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 there might actually exist further, very little known, toolchest MUAs, 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 a feeling of how 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 is a directory tree 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. 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 and were deleted soon. Today, when single messages often include several megabytes of attachments, it 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 Unix’ context, 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 presents 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 of them is small, limited to one function, and has clear boundaries. For the same reasons, they are also good to maintain. Further more, the system can easily get extended. One only needs to put a new program into the toolchest. This was done, for instance, when MIME support was added (e.g. `mhbuild`). Also, different programs can exist to do the basically same job in different ways (e.g. in `nmh`: `show` and `mhshow`).

If someone needs a mail system with some additionally functions that are not available anywhere yet, he best expands a toolchest system like MH. There he can add new functionality by simply adding additional programs to the toolchest. There he does not risk to break existing functionality by doing so.

**Store data in flat text files** is 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 mail messages can operate on the messages in an MH mailbox. All other files MH uses are plain text, too. 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 and without captive user interfaces. If, nonetheless, users want a graphical user interface, they can have it with `xmh` or `exmh`. These are graphical 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.

Providing easy-to-use user interfaces in form of frontends is a good approach, because it does not limit the power of the backend itself. The frontend will anyway 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 hardly be possible. An additional advantage is the possibility 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. Of course, 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 the scripting language may not be very fast, but it can be fast enough though, 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 a toolchest software. Eight tools (two of them have 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 the possibility to only write some parts and though create 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 is few software that so much wants users to tailor their environment and to leverage the use of the software, like MH.

Just to make 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 different format as an additional command, instead of changing the default, he 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 called as scan2.

Make every program a filter is hard to find in MH. The reason therefore is that most of MH’s tools provide basic file system operations for mailboxes. It is the same reason because of which ls, cp, mv, and rm aren’t filters neither. MH does not provide many filters itself, but it is a basis to write filters for. An example would be a mail text highlighter, that means 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 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 problems. There are two main problems: one is technical, the other is about human behavior.

MH is old and email today is very different to email in the time when MH was designed. MH adapted to the changes pretty well, but it is limited, though. MIME support and support for different character encodings is available, but only on a moderate level. This comes from limited development resources. More active developers could quickly change this. But MH is also limited by design, which is the larger problem. IMAP, for example, conflicts with MH’s design to a large extend. These design conflicts are not easily solvable. Possibly, they require a redesign. IMAP may be too different to the classic mail model, which MH covers, so that MH may never support it well.

The other kind of problem are human habits. In this world, where almost all MUAs are monolithic, it is very difficult to convince people to use a toolbox style MUA like MH. The habits are so strong, that even people who understand the concept and advantages of MH do not like to switch, simply because MH is different. Unfortunately, the frontends to MH, which could provide familiar look’n’feel, are quite outdated and thus not very appealing, compared to the modern interfaces of many monolithic MUAs.

5. Case study: uzbl

The last chapter took a look on the MUA MH, which is an old and established 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”; it is pronounced identical.

5.1. Historical background

Uzbl was started by Dieter Plaetinck in April 2009. The idea was born in a thread in the Arch Linux Forums [1]. After some discussion about failures of well known web browsers, Plaetinck (alias Dieter@be) came up with a very sketchy proposal of how a better web browser could look like. To the question of another member, if Plaetinck would write that program, because it would sound 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 an own website [20]. One month after the first code showed up, a mailing list was installed to coordinate and discuss further development, and a wiki was added to store documentation and scripts that showed up on the mailing list and elsewhere.
In the, now, one 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 similar shape.

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, how its name already indicates, the core of uzbl. It handles commands and events to interface other programs, and also displays webpages by using webkit as render engine. Uzbl-browser combines uzbl-core with a bunch of handler scripts, a status bar, an event manager, yanking, pasting, page searching, zooming, and more stuff, to form a “complete” web browser. In the following text, the term “uzbl” usually stands for uzbl-browser, so uzbl-core is included.

Unlike most other web browsers, uzbl is mainly the mediator between the various tools that cover single jobs. Therefore, 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 graphical rendering of the webpage is done by webkit, a web content engine. Uzbl-core is built around libwebkit.

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, so called, handler scripts that wrap external applications which 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, and modules, instead. 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 of 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, of course, highly influenced by existing implementations of web browsers, although they are degenerated. Web browsers should be programs to browse the web, and nothing more. This is the one thing they should do.

Web browsers should not, for instance, manage downloads. This is the job download managers exist for. Download managers do primary care about being good in 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 how could 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. It is covered by the webkit render engine, in uzbl’s case. Audio and video content and files like PostScript, PDF, and the like, are also not the job of a web browser. They should be handled by external applications – ones which’s job is to handle such data. Uzbl strives to do it this way.

Remember Doug McIlroy: “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 matches good here. There was the question, how anyone could want anything less than the best program for the job. But as personal preferences matter, it is probably more important to ask: How could anyone want something else than his preferred program for the job?

Usually users want one program for a specific job. Hence, whenever the task is, for instance, downloading, the same download manager should be used. More advanced users might want to have this download manager in this situation and that one in that situation. They should be able to configure it this way. With uzbl, one can use any download manager the user wants. To switch to a different one, only one line in a small handler script needs to be changed. Alternatively it would be possible to query the program to use by reading a global file or an environment variable, in the handler script.

Uzbl does neither have its own download manager nor depends on a specific one, hence uzbl’s browsing abilities will not be lowered by having a bad download manager. Uzbl’s download capabilities will be just as good as the ones of 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 are the glue in this approach. They make the various parts fit 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.

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, then displays dmenu to let the user select an item, and afterwards 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 extend, actually is a captive user interface. But the difference to most 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. Some are distributed with uzbl, for some external programs, handler scripts are distributed, but arbitrary additional functionality can 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 for humans. Automated programs would crawl the web. That means, they read the source directly. The source includes all the semantics. The graphical representation is just for humans to transfer 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 only to be interactively used by humans. The only case when 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. Modern web browsers have to consist of many thousand lines of code, unfortunately. Using the toolchest approach and wrappers can split the browser into several small parts, tough.

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 a modal interface. Further more, uzbl uses functionality of external tools like wget and netcat. Up to this point, uzbl looks pretty neat and small. The ugly part of uzbl is the web content renderer, webkit. Webkit consists of roughly 400,000 (!) lines of code. Unfortunately, small web render engines are not possible anymore because 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. Development versions of uzbl can be obtained very simply 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. They introduced new features, which were tested for suitability for the main branch. The experiences of 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, too suffers from being different. It is sad, but people use what they know. Fortunately, uzbl’s user interface can look and feel very much the same as 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.

Though, the more important problem is the modern web. The modern web is simply broken. It has state in a state-less protocol, it misuses technologies, and it is helplessly overloaded. The result are web content render engines that must consist of hundreds of thousands lines of code. They also must combine and integrate many different technologies, only to make our modern web accessible. Website to image converter are hardly possible to run without human interaction because of state in sessions, impossible deep-linking, and unautomatable technologies.

The web was misused to provide all kinds of imaginable wishes. Now web browsers, and eventually the users, suffer from it.

### 5.5. Summary

“Uzbl is a browser that adheres to the Unix Philosophy”, that 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 can excellently be seen in uzbl. External tools are used, independent tasks are separated in independent parts and glued together with small handler scripts.

As uzbl, more or less, consists of a set of tools and a bit of glue, anyone can put the parts together and expand it in any desired way. Uzbl is very flexible and customizable. These properties make it valuable for advanced users, but may keep novice users from using it.

But uzbl’s main problem is the modern web, that makes it hard 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 guidelines to good design, in order to create good quality software. Then, real life software, that was designed with the Unix Philosophy in mind, was discussed.

Throughout the paper, the aim was do explain why something should be done the Unix way. It was tried to give reasons that expose that the Unix Philosophy is a preferrable way for designing software.

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

Being able to solve some 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 main enemy.

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

# 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

5. Case study: uzbl ........................................................................................................... 12
   5.1. Historical background ............................................................................................ 12
   5.2. Contrasts to other web browsers .......................................................................... 13
   5.3. Discussion of the design ........................................................................................ 13
   5.4. Problems ................................................................................................................. 15
   5.5. Summary ................................................................................................................ 15

6. Final thoughts ............................................................................................................... 16

References ......................................................................................................................... 17