Empowered End-User Computing:

A Historical Investigation and Development of a File-System-Based Environment

by

Sol Bekic
Student ID 11114279
sol.bekic+ba@s-ol.nu

written in the Winter Term 2019/20
for BA Digital Games
at Cologne Game Lab / TH Köln

supervised by Prof. Dr. Gundolf S. Freyermuth

submitted on 2020-01-07

abstract

Current end-user operating systems are based on a set of design principles and computing paradigms that make them simple to use in some circumstances but are very inflexible for user customization and adaptation. In this thesis, these limitations and design principles will be discussed and contrasted by an analysis of historic systems that solved these issues by following different design goals. Based on this analysis, as well as further literature, an evaluation framework for end-user computing systems is established.

The design and implementation of a new end-user computing system, which focuses on a file system with rich file types and a type coercion system as its central paradigm, is discussed. Following this, the capabilities of the system are demonstrated using multiple example use-cases. An evaluation of these examples as well as the system itself according to the framework established earlier shows that the proposed system is indeed very flexible and useful for a wide variety of uses involving multimedia content from various sources, although the system has many flaws that would hinder widespread adoption.

table of contents

1 introduction

As of December 2019, there are only four major operating systems available to end-users for desktop and mobile devices[1]. All of these systems are comparatively inflexible and can only be customized by users where the system or application developers have anticipated a need with corresponding options. Only a small minority of professional or hobby developers have the knowledge and tools at their disposal to create their own digital experiences, and even for them, the process is often too complex and inefficient to do so effectively.
However, historically, empowered ownership of the computing experience was not meant to be restricted only to end-users. Many computing systems designed during the computer revolution were designed specifically with the goal of complete customizability by users, but little of this approach remains in the systems we use today.

1 N. N. (2019), ***Operating System Market Share - Desktop and Mobile***, https://s-ol.nu/ba/ref/nms

In the first section of this thesis, I will discuss some of the limitations of mainstream operating systems and potential causes in detail. Next, I will highlight some of the contrasting approaches found in historic computing systems, and then derive a framework for the evaluation of end-user computing systems with regard to user empowerment. I will then discuss a new computing system, demonstrate it in some example use cases, and evaluate it using the framework.

2 drawbacks of current systems

The project that this thesis accompanies was created out of frustration with the computing systems that are currently popular and widely available to end-users. The following sections document and exemplify the perceived shortcomings that these systems exhibit, as attributed to specific concepts or paradigms that the systems seem to be designed around.

2.1 application-centric design

The majority of users interact with modern computing systems in the form of smartphones, laptops or desktop PCs, using the mainstream operating systems Apple iOS and Mac OS X, Microsoft Windows or Android.

All of these operating systems share the concept of *Applications* (or *Apps*) as one of the core pieces of their interaction model. The functionality and capabilities of digital devices are bundled, marketed, sold and distributed as applications. This focus on applications as the primary unit of systems can be seen as the root cause of multiple problems.

While there can be a distinction between *Native Applications* and *Web Applications* or *Cloud Services*, the following arguments apply to all different technical implementations of this same concept. The problems associated specifically with Web- and Cloud-based application models will be discussed separately below.

For one, since applications are produced by private companies on the software market, developers compete on features integrated into their applications. To stay relevant, monolithic software or software suites tend to accrete features rather then modularize and delegate to other software[2] . This makes many software packages more complex and unintuitive than they need to be, and also cripples interoperability between applications and data formats.

Because (private) software developers are incentivized to keep customers, they make use of network effects to keep customers locked-in. This often means re-implementing services and functionality that is already available to users, and integrating it directly in other applications or as a new product by the same organization. While this strategy helps big software companies retain customers, it harms users, who have to navigate a complex landscape of multiple incompatible, overlapping, and competing software ecosystems. Data of the same kind, a rich-text document, for example, can be shared easily within the software systems of a certain manufacturer and with other users of the same, but sharing with users of a competing system, even if it has almost the same capabilities, can often be problematic[3].

Another issue is that due to the technical challenges of development in this paradigm, applications are designed and developed by experts in application development, rather than experts in the domain of the tool. While developers may solicit feedback, advice, and ideas from domain experts, communication is a barrier. Additionally, domain experts are generally unfamiliar with the technical possibilities, and may therefore not be able to express feedback that would lead to more significant advances. As a result, creative use of computer technology is limited to programmers, since applications constrain their users to the paths and abilities that the developers anticipated and deemed useful.

The application-centric computing metaphor treats applications as black boxes and provides no means to understand, customize or modify the behavior of apps, intentionally obscuring the inner-workings of applications and completely cutting users off from this type of ownership over their technology. While the trend seems to be to further hide the way desktop operating systems work[4], mobile systems like Apple's *iOS* were designed as such *walled gardens* from the start.

2.2 cloud computing

Web Apps often offer similar functionality to other applications but are subject to additional limitations: In most cases, they are only accessible or functional in the presence of a stable internet connection, and they have very limited access to the resources of the physical computer they are running on. For example, they usually cannot interact directly with the file system, hardware peripherals or other applications, other than through a

2 Chiusano, Paul (2013), ***The future of software, the end of apps, and why UX designers should care about type theory***, http://pchiusano.github.io/2013-05-22/future-of-software.html from 2019-12-18

3 Zielemans, François (2017), ***The Cloud is the Ultimate Vendor Lock-in***, https://www.digital-manifesto.org/the-cloud-is-the-ultimate-vendor-lock-in/ from 2019-11-18

Note that 'creative' here does not only mean 'artistic': this applies to any field, and limits the ability of domain experts to push the boundaries of practice by using technology in innovative ways.

4 N. N. (2014), ***Where is "Show Package Contents"***, https://discussions.apple.com/thread/6740790 from 2019-12-27

standardized set of interactions (e.g. selecting a file via a visual menu, capturing audio and video from a webcam, opening another website).

Cloud software, as well as subscription-model software with online-verification mechanisms, are additionally subject to license changes, updates modifying, restricting or simply removing past functionality, etc. Additionally, many cloud software solutions and ecosystems store the users' data in the cloud, often across national borders, where legal and privacy concerns are intransparently handled by the companies. If a company, for any reason, is unable or unwilling to continue servicing a customer, the user's data may be irrecoverably lost (or access prevented). This can have serious consequences[5], especially for professional users, for whom an inability to access their tools or their cloud-stored data can pose an existential threat.

2.3 inert data (and data formats)

Cragg coins the term "inert data"[6] for the data created and left behind by apps and applications in the computing model that is currently prevalent: Most data today is either intrinsically linked to one specific application, that controls and limits access to the actual information, or, even worse, stored in the cloud where users have no direct access at all. In the latter case, users depend solely on online tools that require a stable network connection and a modern browser and could be modified, removed, or otherwise negatively impacted at any moment.

Aside from being inaccessible to users, the resulting complex proprietary formats are also opaque and useless to other applications and the operating system, which often is a huge missed opportunity: The .docx format, for example, commonly used for storing mostly textual data enriched with images and on occasion videos, is in fact a type of archive that can contain many virtual files internally, such as the various media files contained within. However this is completely unknown to the user and operating system, and so users are unable to access the contents in this way. As a result, editing an image contained in a word document is far from a trivial task: first the document has to be opened in a word processing application, then the image has to be exported from it and saved in its own, temporary file. This file can then be edited and saved back to disk. Once updated, the image may be reimported into the .docx document. If the word-processing application supports this, the old image may be replaced directly, otherwise, the user may have to remove the old image, insert the new one, and carefully ensure that the positioning in the document remains intact.

2.4 disjointed filesystems

5 Lee, Dami (2019, October 7), ***Adobe is cutting off users in Venezuela due to US sanctions***, https://www.theverge.com/2019/10/7/20904030/adobe-venezuela-photoshop-behance-us-sanctions from 2019-12-18

6 Cragg, Duncan (2016), ***Super-powers, but not yours***, http://object.network/manifesto-negative.html from 2019-12-18

The filesystems and file models used on modern computing devices generally operate on the assumption that every individual file stands for itself. Grouping of files in folders is allowed as a convenience for users, but most applications only ever concern themselves with a single file at a time, independent of the context the file is stored in.

Data rarely really fits this concept of individual files very well, and even when it does, it is rarely exposed to the user that way: The ‘Contacts’ app on a mobile phone, for example, does not store each contact’s information in a separate ‘file’ (as the word may suggest initially), but rather keeps all information in a single database file, which is hidden away from the user. Consequently, access to the information contained in the database is only enabled through the contacts application’s graphical interface, and not through other applications that generically operate on files.

Another example illustrates how a more powerful file (organization) system could render such formats and applications obsolete: Given the simple task of collecting and arranging a mixed collection of images, videos, and texts to brainstorm, many might be tempted to open an application like *Microsoft Word* or *Adobe Photoshop* and create a new document there. Both *Photoshop* files and *Word* documents are capable of containing texts and images, but when such content is copied into them from external sources, such as other files on the same computer, or quotes and links from the internet, these relationships are irrevocably lost. As illustrated above, additionally, it becomes a lot harder to edit the content once it is aggregated. To choose an application for this task is a hard trade-off to make, because in applications primarily designed for word processing, arranging content visually is hard to do, and image editing and video embedding options are limited, while tools better suited to these tasks lack nuance when working with text.

To avoid facing this dilemma, a more sensible system could leave the task of positioning and aggregating content of different types to one software component, while multiple different software components could be responsible for editing the individual pieces of content so that the most appropriate one can be chosen for each element. While creating the technological interface between these components is certainly a challenge, the resulting system would greatly benefit from the exponentially-growing capabilities resulting from the modular reuse of components across many contexts: A rich text editor component could be used for example not just in a mixed media collection as proposed above, but also for an email editor or the input fields in a browser.

To summarize, for various reasons, the metaphors and driving concepts of computing interfaces today prevent users from deeply understanding the software they use and the data they own, from customizing and improving their experience and interactions, and from properly owning, contextualizing and connecting their data.

Interestingly, these deficits do not appear throughout the history of today's computing systems but are based in rather recent developments in the field. In fact, the most influential systems in the past aspired to the polar opposites, as I will show in the next section.

3 historical approaches

Two of the earliest holistic computing systems, the Xerox Alto and Xerox Star, both developed at Xerox PARC and introduced in the 70s and early 80s pioneered not only graphical user-interfaces but also the “Desktop Metaphor”. The desktop metaphor presents information as stored in “Documents” that can be organized in folders and on the “Desktop”. It invokes a strong analogy to physical tools. One of the differences between the Xerox Star system and other systems at the time, as well as the systems we use currently, is that the type of data a file represents is directly known to the system.

In a retrospective analysis of the Xerox Star’s impact on the computer industry, the desktop metaphor is described as follows:

> In a Desktop metaphor system, users deal mainly with data files, oblivious to the existence of programs. They do not “invoke a text editor”, they “open a document”. The system knows the type of each file and notifies the relevant application program when one is opened.
>
> The disadvantage of assigning data files to applications is that users sometimes want to operate on a file with a program other than its “assigned” application. […] Star’s designers feel that, for its audience, the advantages of allowing users to forget about programs outweighs this disadvantage.

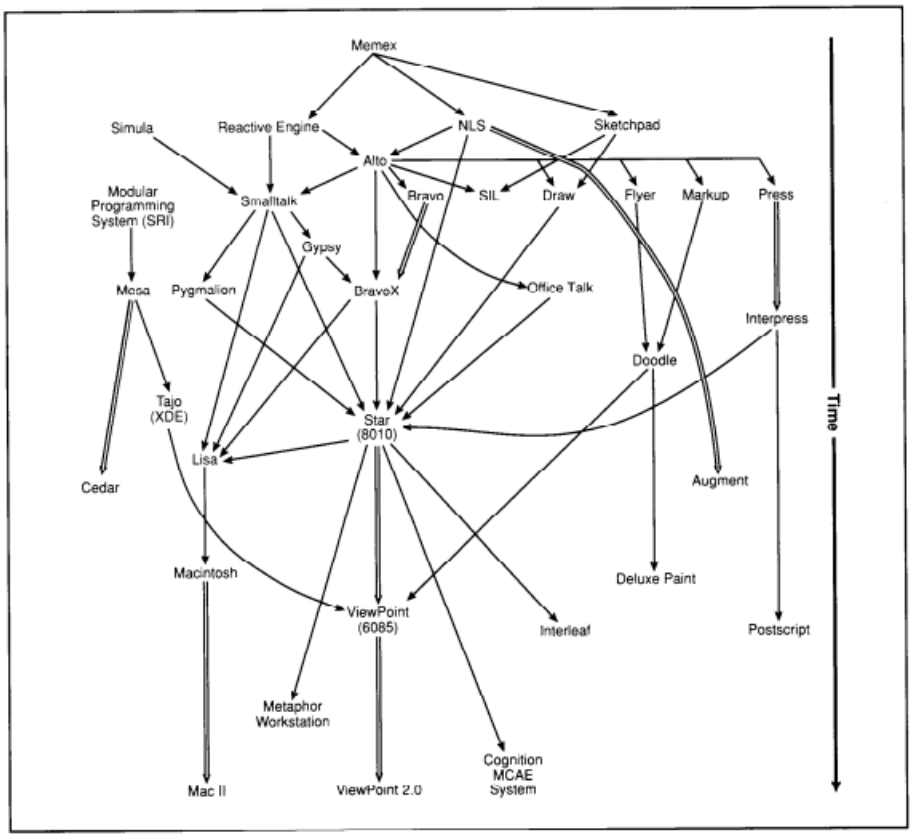
Johnson, Jeff et al. (1989), ***The Xerox Star: A Retrospective***. *Computer*, volume 22, pages 11–26, 28–29. IEEE Computer Society Press. **doi:10.1109/2.35211**

Other systems at the time lacked any knowledge of the type of files, and while mainstream operating systems of today have retro-fit the ability to associate and memorize the preferred applications to use for a given file based on its name suffix, the intention of making applications a secondary, technical detail of working with the computer has surely been lost.

Another design detail of the Star system is the concept of “properties” that are stored for “objects” throughout the system (the objects being anything from files to characters or paragraphs). These typed pieces of information are labeled with a name and persistently stored, providing a mechanism to store metadata such as user preference for ordering or the default view mode of a folder for example.

The earliest indirect influence for the Xerox Alto and many other systems of its time was the *Memex*. The *Memex* is a hypothetical device and system for knowledge management. Proposed by Vannevar Bush in 1945[7] , the concept predates much of the technology that later was used to implement many parts of the vision.

7 Bush, Vannevar (1945), ***As we may think***, https://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/ from 2019-12-18



How systems influenced later systems. This graph summarizes how various systems related to Star have influenced one another over the years. Time progresses downwards. Double arrows indicate direct successors (i.e., follow-on versions). [...]

Johnson, Jeff et al. (1989), ***The Xerox Star: A Retrospective***. *Computer*, volume 22, pages 11–26, 28–29. IEEE Computer Society Press. **doi:10.1109/2.35211**

One of the most innovative elements of Bush's predictions is the idea of technologically cross-referenced and connected information, which would later be known and created as *hypertext*. While hypertext powers the majority of today's internet, many of the advantages that Bush imagined have not carried over into the personal use of computers. There are very few tools for creating personal, highly-interconnected knowledge bases, even though it is technologically feasible and a proven concept (exemplified for example by the massively successful online encyclopedia *Wikipedia*[8]).

8 N. N. (n. d.), ***Hyperlink - Wikis***, https://en.wikipedia.org/wiki/Hyperlink#Wikis from 2019-12-18

While there are few such tools available today, one of the systems that could be said to have come closest to a practical implementation of a Memex-inspired system for personal use might be Apple's *HyperCard*.

In a live demonstration[9], the creators of the software showcase a system of stacks of cards that together implement, amongst others, a calendar (with yearly and weekly views), a list of digital business cards for storing phone numbers and addresses, and a todo list. However these stacks of cards are not just usable by themselves, it is also demonstrated how stacks can link to each other in meaningful ways, such as jumping to the card corresponding to a specific day from the yearly calendar view or automatically looking up the card corresponding to a person's first name from a mention of the name in the text on a different card.

9 N. N. (1987), ***Computer Chronicles, Episode 501***, https://archive.org/details/CC501_hypercard from 2019-12-18

Alongside Spreadsheets, *HyperCard* remains one of the most successful implementations of end-user programming, even today. While its technical abilities have been long matched and surpassed by other software (such as the ubiquitous *Hypertext Markup Language* HTML, and the associated programming language *JavaScript*), these technical successors have failed the legacy of *HyperCard* as an end-user tool: While it is easier than ever to publish content on the web (through various social media and microblogging services), the benefits of hypermedia as a customizable medium for personal management have nearly vanished. End-users do not create hypertext anymore.

4 evaluation framework

In this section, I will collect approaches and reviews of different end-user software systems from current literature, as well as derive and present my own requirements and guiding principles for the development of a new system.

Firstly, I will take a look at a framework for evaluating end-user computing systems from literature, before presenting three concrete design principles and components for a new system.

4.1 qualities of successful end-user computing

Ink and Switch suggest three qualities for tools striving to support end-user programming[10]:

- *Embodiment*, i.e. reifying central concepts of the programming model as more concrete, tangible objects in the digital space (for example, through visual representation), in order to reduce cognitive load on the user.
- *Living System*, by which they seem to describe the malleability of a system or environment, and in particular the ability to make changes at different levels of depth in the system with very short feedback loops and a feeling of direct experience.
- *In-place toolchain*, denoting the availability of tools to customize and author the experience, as well as a certain accessibility of these tools, granted by a conceptual affinity between the use of the tools and general ‘passive’ use of containing system at large.

These serve as guiding principles for the design and evaluation of computer systems for end-users but are by nature very abstract. The following properties are therefore derived as more concrete proposals based on more specific constraints: namely the construction of a system for end-users to keep, structure and display personal information and thoughts.

10 N. N. (2019), ***End-user Programming***, https://inkandswitch.com/end-user-programming.html from 2019-12-18

4.2 modularity

The *UNIX Philosophy*[11] describes the software design paradigm pioneered in the creation of the Unix operating system at the AT&T Bell Labs research center in the 1960s. The concepts are considered quite influential and are still notably applied in the Linux community. Many attempts at summaries exist, but the following includes the pieces that are especially relevant even today:

> Even though the UNIX system introduces a number of innovative programs and techniques, no single program or idea makes it work well. Instead, what makes it effective is the approach to programming, a philosophy of using the computer. Although that philosophy can't be written down in a single sentence, at its heart is the idea that the power of a system comes more from the relationships among programs than from the programs themselves. Many UNIX programs do quite trivial things in isolation, but, combined with other programs, become general and useful tools.
>
> Kernighan, Brian W. and Pike, Rob (1983), ***The UNIX Programming Environment***. Prentice Hall Professional Technical Reference

11 Kernighan, Brian W. and Pike, Rob (1983), ***The UNIX Programming Environment***. Prentice Hall Professional Technical Reference

This approach has multiple benefits with regard to end-user programmability: Assembling the system out of simple, modular pieces means that for any given task a user may want to implement, it is very likely that preexisting parts of the system can help the user realize a solution. Wherever such a preexisting part exists, it pays off designing it in such a way that it is easy to integrate for the user later. Assembling the system as a collection of modular, interacting pieces also enables future growth and customization, since pieces may be swapped out with customized or alternate software at any time.

Settling on a specific modular design model, and reifying other components of a system in terms of it, also corresponds directly to the concept of *Embodiment* described by Ink & Switch.

4.3 content transclusion

The strengths of modular architectures should similarly extend also into the way the system will be used by users. If users are to store their information and customized behavior in such an architecture, then powerful tools need to be present in order to assemble more complex solutions out of such parts. Therefore static content should be able to be linked to (as envisioned for the *Memex*, see above), but also to be *transcluded*, to facilitate the creation of flexible data formats and

The term transclusion refers to the concept of including content from a separate document, possibly stored remotely, by reference rather than by duplication. See also Kolbitsch,

interactions, such that e.g. a slideshow slide can include content in a variety other formats (such as images and text) from anywhere else in the system. Behaviors also should be able to be transcluded and reused to facilitate the creation of ad-hoc systems and applets based on user needs. For example, a user-created todo list should be able to take advantage of a sketching tool the user already has access to.

Josef and Maurer, Hermann (2006), ***Transclusions in an HTML-Based Environment***. *CIT*, volume 14, pages 161-173. **doi:10.2498/cit.2006.02.07**

By forming the immediately user-visible layer of the system out of the same abstractions that the deeper levels of the system are made of, the sense of a *Living System* is also improved: skills that are learned at one (lower) level of the system carry on into further interaction with the system on deeper levels, as does progress in understanding the system's mechanisms.

While there are drawbacks to cloud-storage of data (as outlined above), the utility of distributed systems is acknowledged, and the system should therefore be able to include content and behaviors via the network. This ability should be integrated deeply into the system, so that data can be treated independently of its origin and storage conditions, with as little caveats as possible. In particular, the interactions of remote data access and content transclusion should be paid attention to and taken into consideration for a system's design.

4.4 end-user programming

In order to provide users full access to their information as well as the computational infrastructure, users need to be able to finely customize and reorganize the smallest pieces to suit their own purposes, in other words: be able to program.

While there is an ongoing area of research focusing on the development of new programming paradigms, methodologies, and tools that are more accessible and cater to the wide range of end-users[12], in order to keep the scope of this work appropriate, conventional programming languages are used for the time being. Confidence is placed in the fact that eventually more user-friendly languages will be available and, given the goal of modularity, should be implementable in a straightforward fashion.

12 Edwards, Jonathan (2005), ***Subtext: Uncovering the Simplicity of Programming***. *SIGPLAN Not.*, volume 40, pages 505–518. Association for Computing Machinery. **doi:10.1145/1103845.1094851**

5 system description

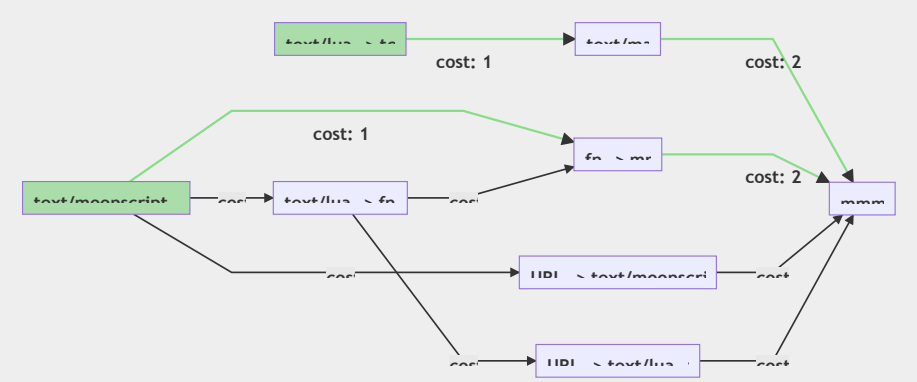
Alongside this thesis, a new end-user computing system has been developed together with the framework presented above. The system, `mmmfs`, is a personal data storage and processing system that was initially developed as a tool for generating static websites but has now been extended with capabilities for live interaction and introspection, as well as embedded editing, as part of this work.

mmmfs has been designed with a focus on data ownership for users. One of the main driving ideas is to unlock data from external data silos and file formats by making data available uniformly across different storage systems and formats. Secondly, computation and interactive elements are also integrated into this paradigm, so that mmmfs can be seamlessly extended and molded to the user's needs.

The abstraction of data types is accomplished using two major components, the *Type System and Coercion Engine* and the *Fileder Unified Data Model* for unified data storage and access. In the next sections, the design and implementation of these two components will be described in detail.

5.1 data storage model

The Fileder Model is the underlying unified data storage model. Like many data storage models it is based fundamentally on the concept of a hierarchical tree-structure.



schematic view of an example tree in a mainstream filesystem

In common filesystems, as pictured, data can be organized hierarchically into *folders* (or *directories*), which serve only as containers of *files*, in which data is actually stored. While *directories* are fully transparent to both system and user (they can be created, browsed, listed and viewed by both), *files* are, from the system perspective, mostly opaque and inert blocks of data.

Some metadata, such as file size and access permissions, is associated with each file, but notably, the type of data is generally not actually stored in the filesystem, but determined loosely based on multiple heuristics depending on the system and context. Some notable mechanisms are:

- Suffixes at the end of the filename are often used to indicate which kind of data a file contains. However there is no centralized standardization of suffixes, and often one suffix is used for multiple incompatible versions of a file-formats, or multiple suffixes are used interchangeably for one format.
- Many file-formats specify a specific data-pattern either at the very beginning or very end of a given file. On UNIX systems the `libmagic` database and library of these so-called *magic constants* is commonly used to guess the file-type based on these fragments of data.
- on UNIX systems, files to be executed are checked by a variety of methods[13] in order to determine which format would fit. For example, script files, the “shebang” symbol, `#!`, can be used to specify the program that should parse this file in the first line of the file.

It should be clear already from this short list that to mainstream operating systems, as well as the applications running on them, the format of a file is almost completely unknown and, at best, educated guesses can be made.

Because these various mechanisms are applied at different times by the operating system and applications, it is possible for files to be labeled or considered as being in different formats at the same time by different components of the system.

This leads to confusion about the factual format of data among users[14], but can also pose a serious security risk: Under some circumstances, it is possible that a file contains maliciously-crafted code and is treated as an executable by one software component, while a security mechanism meant to detect such code determines the same file to be a legitimate image[15] (the file may in fact be valid in both formats).

In mmmfs, the example above might look like this instead:

schematic view of an example mmmfs tree

Superficially, this may look quite similar: there are still only two types of nodes (referred to as *fileders* and *facets*), and again one of them, the *fileders* are used only to hierarchically organize *facets*. Unlike *files*, *facets* don’t only store a freeform *name*, there is also a dedicated *type* field associated with every *facet*, that is explicitly designed to be understood and used by the system.

13 N. N. (2014), ***How does Linux execute a file?*** https://stackoverflow.com/a/23295724/1598293 from 2019-12-18

14 For example, the difference between changing a file extension and converting a file between two formats is often unclear to users, as evident from questions like this: N. N. (2012), ***Why is it possible to convert a file just by renaming it?*** https://askubuntu.com/q/166602 from 2019-12-18

15 Albertini, Ange (2015), ***Abusing file formats; or, Corkami, the Novella***. *PoC||GTFO*, volume 7, pages 18-41

Despite the similarities, the semantics of this system are very different: In mainstream filesystems, each *file* stands for itself only; i.e. in a *directory*, no relationship between *files* is assumed by default, and files are most of the time read or used outside of the context they exist in in the filesystem.

In mmmfs, a *facet* should only ever be considered an aspect of its *fileder*, and never as separate from it. A *fileder* can contain multiple *facets*, but they are meant to be alternate or equivalent representations of the *fileder* itself. Though for some uses it is required, software generally does not have to be directly aware of the *facets* existing within a *fileder*, rather it assumes the presence of content in the representation that it requires, and simple requests it. The *Type Coercion Engine* (see below) will then attempt to satisfy this request based on the *facets* that are in fact present.

Semantically a *fileder*, like a *directory*, also encompasses all the other *fileders* nested within itself (recursively). Since *fileders* are the primary unit of data to be operated upon, *fileder* nesting emerges as a natural way of structuring complex data, both for access by the system and its components, as well as the user themself.

5.2 type system and coercion engine

As mentioned above, *facets* store data alongside its *type*, and when a component of the system requires data from a *fileder*, it has to specify the *expected type* (or a list of these) that it requires the data to be in. The system then attempts to coerce one of the existing facets into the *expected type*, if possible. This process can involve many steps such as converting between similar file formats, running executable code stored in a facet, or fetching remote content. The component that requested the data is isolated from this process and does not have to deal with any of the details.

In the current iteration of the type system, types are simple strings of text and loosely based on MIME-types[16]. MIME types consist of a major- and minor category, and optionally a 'suffix'. Here are some common MIME-types that are also used in mmmfs:

- `text/html` and `text/html+frag` (mmmfs only)
- `text/javascript`
- `image/png`
- `image/jpeg`

While these types allow some amount of specificity, they fall short of describing their content especially in cases where formats overlap: Source code, for example, is often distributed in `.tar.gz` archive files (directory-trees that are first bundled into an `application/x-tar` archive, and then compressed into an `application/gzip` archive). Using either of these two types is respectively incorrect or insufficient information to properly treat and extract the contained data.

To mitigate this problem, mmmfs *types* can be nested. This is denoted in mmmfs *type* strings using the `->` symbol, e.g. the mmmfs *types* `application/gzip -> application/tar -> dirtree` and `URL -> image/jpeg` describe a tar-gz-compressed directory tree and the URL linking to a JPEG-encoded picture respectively.

Depending on the outer type this nesting can mean different things: for URLs, the nested type is expected to be found after fetching the URL with HTTP, compression formats are expected to contain contents of the nested types, and executable formats are expected to output data of the nested type.

It is a lot more important to be able to accurately describe the type of a *facet* in mmmfs than in mainstream operating systems because while in the latter types are mostly used only associate an application that will then prompt the user for further steps if necessary, mmmfs uses the *type* to automatically find one or more programs to execute, in order to convert or transform the data stored in a *facet* into the *type* required in the context where it was requested.

16 Freed, Ned and Klensin, John C. and Hansen, Tony (2013), ***Media Type Specifications and Registration Procedures***. *Request for Comments*, No. 6838. Internet Engineering Task Force. **doi:10.17487/RFC6838**

This process of *type coercion* uses a database of known *converts* that can be applied to data. Every *convert* consists of a description of the input *types* that it can accept, the output *type* it would produce for a given input type, as well as the code for actually converting a given piece of data. Simple *converts* may simply consist of a fixed in and output type, such as this *convert* for rendering Markdown-encoded text to an HTML hypertext fragment:

```
{
  inp: 'text/markdown'
  out: 'text/html+frag'
  transform: (value, ...) ->
    -- implementation stripped for brevity
}
```

Other *converts*, on the other hand, may accept a wide range of input types:

```
{
  inp: 'URL -> image/.*'
  out: 'text/html+frag'
  transform: (url) -> img src: url
}
```

This convert uses a Lua Pattern to specify that it can accept an URL to any type of image, and convert it to an HTML fragment.

By using the pattern substitution syntax provided by the Lua `string.gsub` function, converts can also make the type they return depend on the input type, as is required often when nested types are unpacked:

```
{
  inp: 'application/gzip -> (.*)'
  out: '%1'
  transform: (data) ->
    -- implementation stripped for brevity
}
```

This *convert* accepts an `application/gzip` *type* wrapping any other *type*, and captures that nested type in a pattern group. It then uses the substitution syntax to specify that nested type as the output of the conversion. For an input *type* of `application/gzip -> image/png` this *convert* would therefore generate the type `image/png`.

This last example further demonstrates the flexibility of this approach:

```
{
  inp: 'text/moonscript -> (.*)'
  out: 'text/lua -> %1'
  transform: (code) -> moonscript.to_lua code
}
```

This *convert* transpiles MoonScript source-code into Lua source-code, while keeping the nested type (in this case the result expected when executing either script) the same.

In addition to the attributes shown above, every *convert* is also rated with a *cost* value. The cost value is meant to roughly estimate both the cost (in terms of computing power) of the conversion, as well as the accuracy or immediacy of the conversion. For example, resizing an image to a lower size should have a high cost, because the process is computationally expensive, but also because a smaller image represents the original image to a lesser degree. Similarly, a URL to a piece of content is a less immediate representation than the content itself, so the cost of a *convert* that simply generates the URL to a piece of data should be high even if the process is very cheap to compute.

Cost is defined in this way to make sure that the result of a type-coercion operation reflects the content that was present as accurately as possible. It is also important to prevent some nonsensical results from occurring, such as displaying a link to content instead of the content itself because the link is cheaper to create than completely converting the content does.

Type coercion is implemented using a general pathfinding algorithm, based on *Dijkstra's Algorithm*[17]. First, the set of given *types* is found by selecting all *facets* of the *fileder* that match the *name* given in the query. The set of given *types* is marked in green in the following example graph. From there the algorithm recursively checks whether it can reach other *types* by applying all matching *converts* to the *type* that is the cheapest to reach currently, excluding any *types* that have already been exhaustively-searched in this way. All *types* found that have not yet been inserted into the set of given *types* are then added to the set, so that they may be searched as well.

The algorithm doesn't stop immediately after reaching a *type* from the result set, it continues to search until it either completely exhausts the result-space, or until all non-exhaustively searched paths already have costs higher than the allowed maximum. This ensures that the optimal path is found, even if a more expensive path is found more quickly initially.

excerpt of the graph of conversion paths from two starting facets to mmm/dom

6 example use-cases

17 Dijkstra, Edsger W. (1959), ***A Note on Two Problems in Connexion with Graphs***. *Numer. Math.*, volume 1, pages 269–271. Springer-Verlag. **doi:10.1007/BF01386390**

To illustrate the capabilities of the proposed system, and to compare the results with the framework introduced above, a number of example use cases have been chosen and implemented from the perspective of a user. In the following section I will introduce these use cases and briefly summarize the implementation approach in terms of the capabilities of the proposed system.

The following examples can be viewed and inspected in the interactive version online:

The online version is available at **s-ol.nu/ba**.

- **scripting language support**
- **link to a remote image**
- **markdown content**
- **a gallery of images**
- **a pinwall**

6.1 publishing and blogging

6.1.1 blogging

Blogging is pretty straightforward since it generally just involves publishing lightly-formatted text, interspersed with media such as images and videos or perhaps social media posts. Markdown is a great tool for this job, and has been integrated into the system to much success: There are two different types registered with *converts*: `text/markdown` and `text/markdown+span`. They both render to HTML (and DOM nodes), so they are immediately viewable as part of the system. The only difference for `text/markdown+span` is that it is limited to a single line, and doesn't render as a paragraph but rather just a line of text. This makes it suitable for denoting formatted-text titles and other small strings of text.

The problem of embedding other content together with text comfortably is also solved easily because Markdown allows embedding arbitrary HTML in the document. This made it possible to define a set of pseudo-HTML elements in the Markdown-convert, `<mmm-embed>` and `<mmm-link>`, which respectively embed and link to other content native to mmmfs.

6.1.2 academic publishing

Academic publishing is notoriously complex, involving not only the transclusion of diagrams and other media but generally requiring precise and consistent control over formatting and layout. Some of these complexities are tedious to manage but present good opportunities for programmatic systems and media to do work for the writer.

One of the 'standard' solutions, **LaTeX**, is arguably at least as complex as the mmm system proposed here, but has a much narrower scope, since it does not support interaction.

One such topic is the topic of references. References appear in various formats at multiple positions in an academic document; usually, they are referenced via a reduced visual form within the text of the document and then shown again with full details at the end of the document.

For the sake of this thesis, referencing has been implemented using a subset of the popular BibTeX format for describing citations. Converts have been implemented for the `text/bibtex` type to convert to a full reference format (to `mmm/dom`) and to an inline side-note reference (`mmm/dom+link`) that can be transcluded using the `<mmm-link>` pseudo-tag.

For convenience, a convert from the `URL -> cite/acm` type has been provided to `URL -> text/bibtex`, which generates links to the ACM Digital Library[18] API for accessing BibTeX citations for documents in the library. This means that it is enough to store the link to the ACM DL entry in mmmfs, and the reference will automatically be fetched, and therefore stay up to date with potential remote corrections.

18 N. N. (n. d.), ***ACM Digital Library***, https://dl.acm.org

6.2 pinwall

In many situations, and particularly for creative work, it is often useful to compile resources of different types for reference or inspiration and arrange them spatially so that they can be viewed at a glance or organized into different contexts, etc. Such a pinwall could serve for example to organize references to articles, to collect visual inspiration for a mood board, etc.

As a collection, the Pinwall is primarily mapped to a Fileder in the system. Any content that is placed within can then be rendered by the Pinwall, which can constrain every piece of content to a rectangular piece on its canvas. This is possible through a simple script, e.g. of the type `text/moonscript -> fn -> mmm/dom`, which enumerates the list of children, wraps each in such a rectangular container and outputs the list of containers as DOM elements.

The position and size of each panel are stored in an ad-hoc facet, encoded in the JSON data format: `pinwall_info: text/json`. Such a facet is set on each child and read whenever the script is called to render the children, plugging the values within the facet into the visual styling of the document.

The script can also set event handlers that react to user input while the document is loaded, and allow the user to reposition and resize the individual pinwall items by clicking and dragging on the upper border or lower right-hand corner respectively. Whenever a change is made the event handler can then update the value in the `pinwall_info` facet, so that the updated position and size are stored for the next time the pinwall is opened.

6.3 slideshow

Another common use of digital documents is as aids in a verbal presentation. These often take the form of slideshows, for the creation of which a number of established applications exist. In simple terms, a slideshow is simply a linear series of screen-sized documents, that can be advanced (and rewound) one by one using keypresses.

The implementation of this is rather straightforward as well. The slideshow as a whole becomes a fileder with a script that generates a designated viewport rectangle, as well as a control interface with keys for advancing the active slide. It also allows putting the browser into fullscreen mode to maximize screen space and remove visual elements of the website that may distract from the presentation, and register an event handler for keyboard accelerators for moving through the presentation.

Finally, the script simply embeds the first of its child-fileders into the viewport rectangle. Once the current slide is changed, the next embedded child is simply chosen.

7 evaluation

In this section, I will first take a look at the implementations of the examples for the use cases outlined above, and evaluate them with regard to the framework derived in the corresponding section above. After that, some general concerns and insights that have become apparent while developing the system and working with it will be reviewed.

7.1 examples

7.1.1 publishing and blogging

Since mmmfs has grown out of the need for a versatile content management system for a personal website and blog, it is not surprising to see that it is still up to that job. Nevertheless, it is worth taking a look at its strengths and weaknesses in this context:

The system has proven itself perfect for publishing small- and medium-size articles and blog posts, especially for its ability to flexibly transclude content from any source. This includes diagrams (such as in this thesis), videos (as in the documentation in the appendix), but also less conventional media such as interactive diagrams[19] or twitter postings.

On the other hand, the development of the technical framework for this thesis has posed greater challenges. While the reference and sidenote systems integrated well with the rest of the system, some features like automated table-of-contents and section numbering were less obvious to tackle and finally completed manually. This is mostly due to the approach of splitting up the thesis into a multitude of fileders, and the current lack of mechanisms to re-capture information spread throughout the resulting hierarchy effectively.

19 Bekic, Sol (2019), ***Aspect-ratio independent UIs***, https://s-ol.nu/aspect-ratios from 2019-12-18

7.1.2 pinwall

The pinwall example shows some strengths of the mmmfs system pretty convincingly. The type coercion layer completely abstracts away the complexities of transcluding different types of content, and only positioning and sizing the content, as well as enabling interaction, remain to handle in the pinwall fileder.

A great benefit of the use of mmmfs versus other technology for realizing this example is that the example can seamlessly embed not only plain text, markdown, images, videos, and interactive widgets, but also follow links to all of these types of content, and display them meaningfully. Accomplishing this with traditional frameworks would take great effort, where mmmfs benefits from the reuse of these conversions across the whole system.

In addition, the script for the pinwall folder is 120 lines long, of which 30 lines are styling, while almost 60 lines take care of capturing and handling JS events. The bulk of complexity is therefore shifted towards interacting with the UI layer (in this case the browser), which could feasibly be simplified through a custom abstraction layer or the use of output means other than the web.

7.1.3 slideshow

A simplified image slideshow example consists of only 20 lines of code and demonstrates how the reactive component framework simplifies the generation of ad-hoc UI dramatically:

```
import ReactiveVar, text, elements from require 'mmm.component'
import div, a, img from elements

=>
  index = ReactiveVar 1

  prev = (i) -> math.max 1, i - 1
  next = (i) -> math.min #@children, i + 1

  div {
    div {
      a 'prev', href: '#', onclick: -> index\transform prev
      index\map (i) -> text " image ##{i} "
      a 'next', href: '#', onclick: -> index\transform next
    },
    index\map (i) ->
      child = assert @children[i], "image not found!"
      img src: @children[i]\gett 'URL -> image/png'
  }
```

The presentation framework is a bit longer, but the added complexity is again required to deal with browser quirks, such as the fullscreen API and sizing content proportionally to the viewport size. The parts of the code dealing with the content are essentially identical, except that content is transcluded via the more general `mmm/dom` type-interface, allowing for a greater variety of types of content to be used as slides.

7.2 general concerns

While the system has proven pretty successful and moldable to the different use-cases that it has been tested in, there are also limitations in the proposed system that have become obvious in developing and working with the system. In this section, these limitations will be discussed individually, and directions for further research and solutions will be given where apparent.

7.2.1 global set of converts

In the current system, there is only a single, global set of *converts* that can be potentially applied to facets anywhere in the system. Therefore it is necessary to encode behavior directly (as code) in facets wherever exceptional behavior is required. For example, if a fileder containing multiple images wants to provide custom UI for each image when viewed independently, this code has to either be attached to every image

individually (and redundantly), or added as a global convert. To make sure this convert does not interfere with images elsewhere in the system, it would be necessary to introduce a new type and change the images to use it, which may present even more problems, and works against the principle of compatibility that the system has been constructed for.

A potential direction of research in the future is to allow specifying *converts* as part of the fileder tree. Application of *converts* could then be scoped to their fileders' subtrees, such that for any facet only the *converts* stored in the chain of its parents upwards are considered. This way, *converts* can be added locally if they only make sense within a given context. Additionally, it could be made possible to use this mechanism to locally override *converts* inherited from further up in the tree, for example, to specialize types based on their context in the system.

The biggest downside to this approach would be that it presents another pressure factor for, while also reinforcing, the hierarchical organization of data, thereby exacerbating the limits of hierarchical structures.

See also Jacobs, B. (2004), ***Alternatives to Trees***, https://www.oocities.org/tablizer/sets1.htm from 2019-12-18

7.2.2 code outside of the system

At the moment, a large part of the mmmfs codebase is still separate from the content and developed outside of mmmfs itself. This is a result of the development process of mmmfs and was necessary to start the project as the filesystem itself matured, but has now become a limitation of the user experience: potential users of mmmfs would generally start by becoming familiar with the operation of mmmfs from within the system, as this is the expected (and designated) experience developed for them. All of the code that lives outside of the mmmfs tree is therefore invisible and opaque to them, actively limiting their understanding, and thereby the customizability, of the system.

This weakness represents a failure to (fully) implement the quality of a "Living System" as proposed by *Ink and Switch*[20].

20 N. N. (2019), ***End-user Programming***, https://inkandswitch.com/end-user-programming.html from 2019-12-18

In general, however, some portion of code may always have to be left outside of the system. This also wouldn't necessarily represent a problem, but in this case it is particularly relevant for the global set of *converts* (see above), as well as the layout used to render the web view. Both of these are expected to undergo changes as users adapt the system to their own content types and domains of interest, as well as their visual identity, respectively.

7.2.3 type system

The currently used type system based on strings and pattern matching has been largely satisfactory but has proven problematic for some anticipated use cases. It should be considered to switch to a more intricate, structural type system that allows encoding more concrete meta-data alongside the type, and to match *converts* based on a more flexible scheme of pattern matching. For example, it is envisaged to store the resolution of an image file in its type. Many *converts* might choose to ignore this additional information, but others could use this information to generate lower-resolution ‘thumbnails’ of images automatically. Using these mechanisms for example images could be requested with a maximum-resolution constraint to save on bandwidth when embedded in other documents.

7.2.4 type-coercion

By giving the system more information about the data it is dealing with, and then relying on the system to automatically transform between data-types, it is easy to lose track of which format data is concretely stored in. In much the same way that the application-centric paradigm alienates users from an understanding and feeling of ownership of their data by overemphasizing the tools in between, the automagical coercion of data types introduces distance between the user and an understanding of the data in the system. This poses a threat to the transparency of the system, and potentially a lack of the “Embodiment” quality (see above).

Potential solutions could be to communicate the conversion path clearly and explicitly together with the content, as well as making this display interactive to encourage experimentation with custom conversion queries. Emphasizing the conversion process more strongly in this way might be a way to turn this feature from an opaque hindrance into a transparent tool. This should represent a challenge mostly in terms of UX and UI design.

7.2.5 in-system editing

Because many *converts* are not necessarily reversible, it is very hard to implement generic ways of editing stored data in the same format it is viewed. For example, the system trivially converts markdown-formatted text sources into viewable HTML markup, but it is hardly possible to propagate changes to the viewable HTML back to the markdown source. This particular instance of the problem might be solvable using a Rich-Text editor, but the general problem worsens when the conversion path becomes more complex: If the markdown source was fetched via HTTP from a remote URL (e.g. if the facet’s type was `URL -> text/markdown`), it is not possible to edit the content at all, since the only data owned by the system is the URL string itself, which is not part of the viewable representation. Similarly, when viewing output that is generated by code (e.g. `text/moonscript -> mmm/dom`), the code itself is not visible to the user, and if the user wishes to change parts of the representation, the system is unable to relate these changes to elements of the code or assist the user in doing so.

However, even where plain text is used and edited, a shortcoming of the current approach to editing is evident: The content editor is wholly separate from the visible representation, and only facets of the currently viewed fileder can be edited. This means that content cannot be edited in its context, which is exacerbated by the extreme fragmentation of content that mmmfs encourages.

As a result, interacting with the system at large is still a very different experience from editing content (and thereby extending the system) in it. This is expected to represent a major hurdle for users getting started with the system and is a major shortcoming in enabling end-user programming, as set as a goal for this project. A future iteration should carefully reconsider how editing could be integrated more holistically with the other core concepts of the design.

7.2.6 end-user adoption

As mentioned above, a conscious choice was made to exclude the implementation of a dedicated end-user programming facility in the system, and instead conventional programming languages and mechanisms were relied upon as the central way of customizing the system and experience. While this was a crucial choice to make in order to proceed with the project as a whole, it means that the system currently can not be adopted and used to its full extent by end-users. This also means that a full evaluation of the system with regard to end-user empowerment has to be left open until this can be changed by further work.

8 conclusion

The historical analysis and the evaluation of the proposed system show that many of the limitations of current mainstream operating systems can be worked around effectively. The flaws can also be attributed in part to some concrete design paradigms, which future system designers may seek to avoid. To this end, the framework provided for evaluation may also be useful.

The system proposed and developed in the project corresponding to this thesis has been shown to successfully implement many of the properties were hoped to be achieved, such as a modular and consistent architecture and strong support for mixed-content transclusions. On the other hand, some limitations in the design are apparent. Many of these limitations constitute candidate topics for further research, and most can be attributed to trade-offs made in the development process.

references

1. Albertini, Ange (2015), ***Abusing file formats; or, Corkami, the Novella***. *PoC||GTFO*, volume 7, pages 18-41
2. Bekic, Sol (2019), ***Aspect-ratio independent UIs***, https://s-ol.nu/aspect-ratios from 2019-12-18
3. Bush, Vannevar (1945), ***As we may think***, https://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/ from 2019-12-18
4. Chiusano, Paul (2013), ***The future of software, the end of apps, and why UX designers should care about type theory***, http://pchiusano.github.io/2013-05-22/future-of-software.html from 2019-12-18
5. Cragg, Duncan (2016), ***Super-powers, but not yours***, http://object.network/manifesto-negative.html from 2019-12-18
6. Dijkstra, Edsger W. (1959), ***A Note on Two Problems in Connexion with Graphs***. *Numer. Math.*, volume 1, pages 269–271. Springer-Verlag. **doi:10.1007/BF01386390**
7. Edwards, Jonathan (2005), ***Subtext: Uncovering the Simplicity of Programming***. *SIGPLAN Not.*, volume 40, pages 505–518. Association for Computing Machinery. **doi:10.1145/1103845.1094851**
8. Freed, Ned and Klensin, John C. and Hansen, Tony (2013), ***Media Type Specifications and Registration Procedures***. *Request for Comments*, No. 6838. Internet Engineering Task Force. **doi:10.17487/RFC6838**
9. Jacobs, B. (2004), ***Alternatives to Trees***, https://www.oocities.org/tablizer/sets1.htm from 2019-12-18
10. Johnson, Jeff et al. (1989), ***The Xerox Star: A Retrospective***. *Computer*, volume 22, pages 11–26, 28–29. IEEE Computer Society Press. **doi:10.1109/2.35211**
11. Kernighan, Brian W. and Pike, Rob (1983), ***The UNIX Programming Environment***. Prentice Hall Professional Technical Reference
12. Kolbitsch, Josef and Maurer, Hermann (2006), ***Transclusions in an HTML-Based Environment***. *CIT*, volume 14, pages 161-173. **doi:10.2498/cit.2006.02.07**
13. Lee, Dami (2019, October 7), ***Adobe is cutting off users in Venezuela due to US sanctions***, https://www.theverge.com/2019/10/7/20904030/adobe-venezuela-photoshop-behance-us-sanctions from 2019-12-18
14. N. N. (n. d.), ***ACM Digital Library***, https://dl.acm.org
15. N. N. (1987), ***Computer Chronicles, Episode 501***, https://archive.org/details/CC501_hypercard from 2019-12-18
16. N. N. (2019), ***End-user Programming***, https://inkandswitch.com/end-user-programming.html from 2019-12-18
17. N. N. (2014), ***How does Linux execute a file?*** https://stackoverflow.com/a/23295724/1598293 from 2019-12-18
18. N. N. (n. d.), ***Hyperlink - Wikis***, https://en.wikipedia.org/wiki/Hyperlink#Wikis from 2019-12-18
19. N. N. (2019), ***Operating System Market Share - Desktop and Mobile***, https://s-ol.nu/ba/ref/nms
20. N. N. (2014), ***Where is "Show Package Contents"***, https://discussions.apple.com/thread/6740790 from 2019-12-27
21. N. N. (2012), ***Why is it possible to convert a file just by renaming it?*** https://askubuntu.com/q/166602 from 2019-12-18
22. Zielemans, François (2017), ***The Cloud is the Ultimate Vendor Lock-in***, https://www.digital-manifesto.org/the-cloud-is-the-ultimate-vendor-lock-in/ from 2019-11-18

appendix: project log

The following pages document the development of the `mmmfs` system described above in the form of a project log. Please note that the log has been written primarily for viewing using a web browser, and as such makes extensive use of hyperlinking, and also includes some videos that cannot be reproduced in print. It is therefore recommended to view the live version of the log online at the following address: **s-ol.nu/ba/log**.

statement of originality

This is to certify that the content of this project, documentation and thesis is my own work. It has not been submitted for any other degree or other purposes. I certify that the intellectual content of my submission is the product of my own work and that all the assistance received in preparing it as well as all sources used have been properly acknowledged.

Sol Bekic