More Unixy Than Unix

A Text-Oriented Programmable Environment

By Francesc Hervada-Sala (mail: francesc at the-text.net)

Published Oct 26, 2016.

URL: http://the-text.net/more-unixy-than-unix

Contents

Unix implemented a brilliant design principle several decades ago. Could this principle be applied consistently in all kinds of software, from server applications like databases to rich client applications? Yes, provided we deepen in the concept of text and redefine it as a parsed structure instead of as a human readable string. This will lead to software applications that are simpler to build, easier to use, and much more powerful than current ones.

Unix: The Design Principle

Characteristics of the Unix Environment

Let us bring into mind the design principle behind the Unix operating system.

There is a file system which provides access to all entities the system is aware of from a unified name space. Files contain raw strings for representing both data and source code, this way generic file editors —be it interactive such as vi or command-based such as sed— can manipulate all data and all source code in the system.

Programs are filters that get a string stream as input and generate a string stream as output. That makes it possible to combine each program with any other one. Strings can be be passed to a program as arguments when calling it in order to control its behavior. Additionally, program configuration parameters are held in raw string files, so that any generic file editor —interactive or automatic— can regulate all aspects of the system's functionality.

The shell is an interactive environment in which the user can edit commands and execute them. It uses a simple grammar that provides uniform access to all installed functionality. Also the language used is extensible —via aliases and function definitions—.

The key concept in Unix is the concept of a string stream. A string stream is a sequence of human readable and writable sequences of characters. The system supports sequences of character sequences; users represent their data and code as character sequences using multiple notations, formats and languages. To use the system, a user builds a command and instructs the system to execute it. The system provides generic tools that the user combines and applies to her particular needs. If the available commands for the application are well-thought-out —each of them is simple and orthogonal to each other—, a proficient and smart user has a powerful system at her fingertips.

Current Software Landscape

Yet when you look at current software, you realize that even Unix installations are not unixy at all. There are lots of components like databases, office applications, non line-oriented file formats such as XML and many more things that do not integrate well with the described principle. You cannot use vi, sed, grep, cp, rsync or xargs with them. For example, you cannot rsync a table in MariaDB to replicate it, you cannot grep inside a LibreOffice document, you cannot vi a PDF document.

Current mainstream software is based on applications. Each application has its own data, provides access to users through a proprietary UI and to applications through a proprietary API and uses a proprietary file format for exchanging data through static files. Web applications are an extreme case of this: even data storage and application services are hosted on the publisher's servers.

The key concept in mainstream software is the concept of the standalone application. The system contains some installed applications, which are responsible for particular tasks and the sole owners of particular data. For a particular job, the user must choose an application and use its built-in functionality. The user is treated like a little child whom a mentor must provide everything, like a prisoner that cannot freely decide where to go. In contrast, in a unixy environment the user feels like an adult, free citizen.

Why didn't the Unix Principle take over the world?

The Unix Principle is superior, but if it is really such a better approach, why didn't it take over the world? Why don't all software systems currently work that way?

I posit that there are some implementation constraints in Unix —which was conceived over forty years ago— that impede the thorough use of this approach for all purposes.

Unix presents some scalability issues. Static files work well for many purposes, but cannot be generalized for everything. You cannot use static files to store millions of data records accessed concurrently by hundreds of users, you wouldn't be able to manage them and query them with good performance. Also the fact that every command runs in a dedicated process poses a limitation. You cannot use this model for small functions that are used repeatedly during the program execution. Scalability is also constrained by the lack of strong referencing. The names of entities are represented by strings in files and resolved at run time, therefore references brake when files are renamed or deleted.

Unix also suffers from usability problems, because only power users feel comfortable with its user interface.

There are two flaws regarding these implementation constraints. While Unix hits the mark by using text as a universal interface, it fails by reducing text to a string representation. It also fails by coupling interfaces to their implementation (i.e. commands, pipes, streams are always implemented the same way).

Let's now take a look at some experiments I've been working on to overcome these flaws.

UText: The Precursor

The Universal-Text Interpreter

The Universal-Text Interpreter is a file generation tool that I wrote some years ago with Perl. It provides a shell with which you can generate files in multiple formats (i.e. HTML, LaTeX, OpenDocument, PDF) from a set of source files in Universal-Text Language (UTL) or custom formats.

To use the tool, you declare a data structure, then you enter data, and last you define a transformation script. For example, you can declare a data structure this way to represent websites that consist of pages, each page having a title and a body that contains heading and paragraph strings:

```
^ website {
    ^ page {
        ^ title : string
        ^ body {
            ^ tag : string
            ^ h1 : tag
            ^ p : tag
        }
    }
}
```

Having declared a data structure, you can enter some data:

```
=web1 ~website
=index ~page
~title Welcome
~body
~h1 Welcome to my great new site!
~p You may ask what is this site about.
~p Well, I am asking myself that, too!
```

Then you define a transformation script that determines how to generate HTML files:

```
select website.page begin
    save [u].html begin
        out <html><head><title>[v title]</title></head><body>
        select body.:tag do out <[u.role]>[v]</[u.role]>
        out </body></html>
    end
end
```

When you run the script, the interpreter visits all pages inside websites and generates, for each page, a file with its name and the extension .html. Each file contains, besides a header and a footer, a line for each tag (heading or paragraph) inside the body of the page expressed as an HTML tag. Usage example:

```
$ ut -b web1.utl
Creating file index.html
$ cat index.html
<html><head><title>Welcome to my great new site!</title></head><body>
<h1>Welcome</h1>
<p>Welcome to my new site!</p>
<p>You may ask what is this site about.</p>
<p>Well, I am asking myself that, too!</p>
</body></html>
```

UText Samples

This is the beginning of a real blog post written with UTL. Prose is entered unformatted, as regular writing, and there are just some structural marks preceded with a tilde or an equal sign.

```
~post =computer-text-machine
~title Computer = Text Machine
~timestamp 2014-08-16 16:01:50 CET
~content

Given that computers are capable of many different things, it is
easy to get the impression that they are universal machines.
Yet this cliché is absolutely wrong. Let us take a closer look
at what computers really are.

Software systems are typically divided into applications. An
application is used for word processing, another for email,
and others to respectively manage a spreadsheet, browse the Web,
or edit a photo.
```

The Universal-Text Interpreter can also be used for software development. For example, a C module description could look similar to this:

```
=text ~module
~title Text
~file text.c
~description This module implements basic services to
    define and query text units.
```

This describes a C code module implemented in the source file `text.c`. Then you declare which other modules this module depends on.

```
~uses base
~uses transaction
~uses event
```

Last you define some constants that this module exposes.

```
~error E_UNIT
    ~caption Unknown text unit
    ~description There is no text unit with the given Id.
~error E_TYPE
    ~caption Incorrect type
    ~description The given type does not correspond to the unit role.
```

When you run the Universal-Text Interpreter, it generates a website with the documentation of all the code modules and it generates also some source code. For example, a file `text.i`, the only file to be included in `text.c`, that includes all modules needed:

```
$ cat text.i
    #include "base.h"
    #include "transaction.h"
    #include "event.h"
    #include "constants.h"
```

It generates the makefile:

```
[...]
text.o: text.c base.h transaction.h event.h constants.h
[...]
```

Also a module implementing the constants is automatically generated:

```
$ cat constants.h
    char * str_constant(int c);
    #define E_UNIT 1 //Unknown text unit
    #define E_TYPE 2 //Incorrect type

$ cat constants.c
    char * str_constant(int c) {
        switch(c) {
            case E_UNIT: return "Unknown text unit";
            case E_TYPE: return "Incorrect type";
        }
    }
```

With this approach you generate documentation and code from the same source. This ensures that the documentation is accurate and up to date, and you save programming and debugging effort because parts of the code are automatically generated.

We've seen the general notion of the Universal-Text Interpreter. Text is used here as a universal data structure, and you generate strings by transforming text units through a script language. Regular use of it for many purposes has shown that it can represent all kinds of data. Data structures are easily defined and can be flexibly changed at any time, without requiring much rework of the source files.

Now let's consider the Text Engine, which is the next step in my exploration.

Text Engine: The Experiment

I am currently developing the Text Engine. It is a user space tool written in C. It will provide a shell with which text can be parsed, queried, transformed and generated. Yet text is here implemented not as a string, but as structured data.

Data Representation: Symbolic Language

The Text Engine supports a symbolic language that can represent any text as a set of symbols with cross-references between them. Data is implemented as text. For example, you can define a data structure for to-do lists with this symbolic language expression:

```
~=to-do list {
  ~=task {
    ~=description :string;
    ~=due :date;
    ~=done :boolean;
  }
};
```

The above defines a text unit named «to-do list» that contains text units for single tasks, each of which has a description —which is a text unit of type «string»—, a due date and a flag indicating whether the task has been completed. Once you have defined a unit, you can instantiate it. You can enter data for your task list with the following expression:

```
~to-do list =conference work {
   ~task {
     ~description "book airplane ticket";
     ~due "09/14/2015";
   };
   ~task {
     ~description "book accommodation";
     ~due "09/28/2015";
     ~done #true;
   };
   ~task {
     ~description "travel";
     ~due "10/14/2015";
   };
};
```

This defines three tasks for the to-do list with name «conference work», each with a description and a due date.

Code Representation: Script Language

The Text Engine provides a script language designed to represent code. To show all tasks of all to-do lists, you can execute this:

```
select ~to-do list . task;
```

To get all tasks of the to-do list «conference work» that are not completed, run this:

```
select =conference work . ~task (not having ~done);
```

You can specify the output of a select statement. For example, if you want to get a line with the description for each task, you do:

```
select ~to-do list . task (order by ~due) {
    println ~description;
}
```

Or, to output the description and due date of each task:

```
select ~to-do list . task (order by ~due) {
    println "[v ~due] [v ~description]";
}
```

In the Text Engine, code is text. Any script language expression is represented internally as a text unit —its parse tree— and can therefore be expressed with symbolic language. For example, the last instruction is equivalent to the following expression in symbolic language:

```
~transformation :select {
    ~selector {
        ~level { ~target role #to-do list };
        ~level { ~target role #task };
    }
    ~order by {
        ~target role #due;
    }
    ~body {
        ~transformation :println {
            ~segment :variable { ~role #due };
            ~segment :literal { #space };
            ~segment :variable { ~role #description };
        }
    }
}
```

Text Engine v. 1.0 — Applications

The Text Engine will consist of a base layer —providing support for symbolic and script language and basic functionality— and some modules that will be used for several applications.

An application that will be available from the very beginning will be file generation. Basic functions such as `select` to query the repository, `print` to output a string and `save` to store a string as a file will allow the user to export text units as files. For example, the following script generates a file containing all to-do lists:

```
save "./todo.txt" {
    select ~to-do list {
        println "=== [u] ===";
        select ~task (order by ~due) {
            print "[v ~due] [v ~description]";
            if ~done { print " (done)" };
            println;
        }
    }
}
```

When running the above, the Text Engine creates a file named «todo.txt» with this content:

```
=== conference work ===
09/14/2015 book airplane ticket
09/28/2015 book accommodation (done)
10/14/2015 travel
```

To work with files of particular formats you proceed in the following way. You define a unit that represents the file format semantics and serializes and deserializes their instances and then you write scripts to convert your repository data to such semantics.

For example, to work with LibreOffice Writer files, you would use a module that defines a unit named `odt` similar to this:

```
~=odt {
    ~=title :string;
    ~=content {
        ~=chapter {
            ~=heading :string;
            ~=paragraph :string;
        }
    ~=save :transformation;
    ~=read :transformation;
    }
}
```

This defines an OpenDocument unit with a title and content consisting of chapters with a heading and some paragraphs. The function `save` receives a unit of type `odt` as input and serializes it into a file with LibreOffice format. The function `read` receives a file as input and generates a unit of type `odt` in the text repository.

For example, a simple document looks like this:

```
:odt {
    ~title "My Book";
    ~content {
        ~chapter {
            ~heading "Prolog";
            ~paragraph "This book is unlike any other book.";
        }
    }
}
```

If you want to export a to-do list as a LibreOffice Writer file, you execute this script:

```
save :odt "./todo.odt" {
    out ~title "To Do";
    out ~content {
        select ~to-do list {
            out ~chapter {
                out ~heading [u];
                select ~task (order by ~due) {
                    out ~paragraph "[v ~due] [v ~description]";
                }
            }
        }
    }
}
```

The above generates the following text unit and stores it as an Open Document file:

```
:odt {
    ~title "To Do";
    ~content {
       ~chapter {
          ~heading "conference work";
          ~paragraph "09/14/2015 book airplane ticket":
          ~paragraph "09/28/2015 book accommodation";
          ~paragraph "10/14/2015 travel";
        }
    }
}
```

To import a to-do list from a LibreOffice Writer file, you execute this:

```
read :odt "./todo.odt" {
    select ~content.chapter {
        out ~to-do =[v heading] {
            select ~paragraph {
                out ~task [inline/][v][/inline];
            }
        }
    }
}
```

This way you can work with conventional applications on data that is stored in the Text Engine's repository. You can use several applications on the same data. For example, if you are writing a book, you can work on the book's outline with an outliner tool, edit some or all chapters with a word processor and manage the writing process with a spreadsheet. As all data comes from a single source, changes made in one application apply automatically to the other ones.

The next step will be to implement support for virtual files in the Text Engine. Then, import and export functions will be executed automatically in the background by the Text Engine whenever required, which will simplify usage and produce a smoother user experience.

Text Engine v. X.0 — Graphical UI

When there is a demand for it, a graphical user interface based on the Text Engine can be built representing graphics as text. For example, you open a window executing the following:

```
open :window {
    ~title "Window 1";
    ~client area : stack panel {
        ~panel :string panel {
            ~content "Window 1";
            ~format #heading1;
        }
        ~panel :string panel {
            ~content "This is the first window.";
        }
    }
}
```

The above code creates a window on the screen with two lines, the first one formatted as a header:

```
== Window 1 ==
This is the first window.
```

To show the to-do list one can execute this script expression:

```
open :window {
    select =conference work {
        out ~title [u];
        out ~client area : list panel {
            select ~task (order by ~due) {
                out ~panel :string panel "[v ~due] [v ~description]";
            }
        }
    }
}
```

Such a window would look like this:

```
== conference work ==
* 09/14/2015 book airplane ticket
* 09/28/2015 book accommodation
* 10/14/2015 travel
```

Also UI controls will be represented by text units. For example, this defines a menu item:

```
~window {
    ~menu bar {
        ~menu item {
            ~caption "Insert Date";
            ~action "select #cursor position { insert ~date {#now}}";
        }
    }
}
```

If you are working with your to-do list and want to add a menu option to store the current data as a raw string file, you can execute this:

```
put #window1.~menu bar {
  ~menu item {
    ~caption "To raw string file";
    ~action [q/]
        save "[u #data source :to-do list].txt" {
          select ~task {
            println "[v due] [v description]";
          }
        }
        [/q];
    }
  }
};
```

After that you see a new item «To raw string file» at the menu bar. When you activate it, a file with the name of the to-do list and extension «txt» is generated containing a list of all tasks.

Text-Oriented Software

Text Engine — The Principle

We have seen that with the Text Engine one can build a software system based on text. Data structures are text units, code procedures are text units, so are UI elements and everything else. The Text Engine provides support for querying and transforming text units, and thus it provides means to handle all data, all code, all UI elements and everything else in the system in a uniform way. You don't use a proprietary interface to access each system component, you use a generic interface instead, namely text.

It is important to stress that this does not imply that there is a single system-wide implementation of text. Each software piece can implement its own text units internally in a proprietary way. Only the communication between software pieces takes place through a generic data structure that represents text units.

TextOS: The Dream

One can dream of an operating system completely built upon this principle. Let us call it the «Text Operating System», TextOS. The kernel has a text engine. There is a system-wide (virtual) text repository that contains all data and all software. There are no files at all.

Let us see how a TextOS would look and feel comparing it with Unix. In Unix, you get a list of all your document names by running:

```
ls documents
```

In TextOS, you can also get all document names with:

```
ls ~document;
```

Yet you can also make a much more specific query, for example get all documents that must be completed by next week:

```
ls ~document (having ~due date (lt #next week));
```

You can also specify what output you want to get, for example each document's title followed by the status in brackets:

```
ls ~document (order by #modify date) {
    println "[v title] ([v status])";
};
```

If you want to receive an email every day with your open tasks, in Unix you proceed in the following way. First, you create a file:

```
vi generate-open-tasks.pl
```

You write a script in a programming language that retrieves your open tasks and outputs them. After that, you mark the file as executable:

```
chmod u+x generate-open-tasks.pl
```

Then you add a line at a configuration file in order to instruct the cron daemon to run a job daily:

```
echo "0 6 * * *
   mary /home/mary/generate-open-tasks.pl
   | mail -s tasks mary@example.com
   " >>/etc/crontab
```

In TextOS, you achieve the same by running a single script instruction:

```
cron #daily {
  mail "mary@example.com" (subject "tasks") {
      select :task (not having ~done; order by ~due) {
          println ~description;
      }
  }
};
```

Note that above a single language has access to several components. Additionally, it is not mandatory to use this particular language, and other languages (i.e. symbolic language) would also do. In contrast, in Unix to do the same you must use several notations and languages (Perl, bash, cron notation) that you cannot choose.

In Unix, you can see what jobs are scheduled by running `cat`:

```
cat /etc/crontab
    0 6 * * * root logrotate /etc/logrotate.conf
```

In TextOS, you can do the same:

```
cat #etc.crontab;
    0 6 * * * root logrotate /etc/logrotate.conf
```

Yet you can also get the scheduled jobs shown in alternative formats, for example as a symbolic language expression:

```
cat :symbol #etc.crontab;
    ~job {
        ~time {
            ~month #all;
            ~day #all;
            ~wday #all;
            ~hour "6";
            ~minute "0";
        };
        ~user #root;
        ~command #logrotate {
            ~configuration #etc.logrotate.conf
        }
    };
```

In TextOS, you can express everything in alternative formats, which can be useful if you are unfamiliar with a particular notation. Generic languages such as symbolic language do not substitute for specific languages, but complement them. No language is imposed, the user can always choose.

The TextOS will also avoid a lot of redundant implementation of the same functionality. For example, in Unix there are multiple implementations of the mail functionality. There is a mail shell command, there are many Perl modules at CPAN that implement mail for using it in Perl programs, and the same functionality is implemented in several modules for say PHP. In TextOS a single interface for mail would suffice, for example:

```
~=mail {
    ~=from :string;
    ~=to :string;
    ~=subject :string;
    ~=body { [...] };
    ~=send :transformation;
};
```

Whenever you need to send an email, you instantiate a unit of this type and execute a `send` operation on it. Different programming languages will of course be able to express the creation and sending of

email with different syntax, but the compiler will not have to implement the function itself as in today's systems, it will just refer to the generic interface.

Today, every mail implementation must be configured differently. You must put particular files under `/etc` for the mail shell command and set particular variable values in your Perl or your PHP script to control the mail functionality. The same information such as sender name, address and the SMTP server name must be defined more than once at several places and with distinct notations. TextOS will support a system-wide single configuration. And the user will be able to choose what notation to use for seeing and editing it.

In TextOS there can be obviously more than one implementation of the mail functionality available. Then, the system's administrator of the TextOS instance will be able to choose where what implementation of the mail functionality has to be used, and she will be able to change that at run time —manually or programmatically—, without having to change the configuration parameters at all.

In Unix, file content is not accessible to the operating system and file systems provide some limited metadata such as permissions and timestamps. If applications need some more metadata, they have to implement it themselves. This way, you need special file formats for metadata of particular applications, for example in current systems you use ID3 metadata for MP3 audio files and Exif metadata for JPEG image files.

In TextOS, there is, as a matter of principle, no need for a «metadata» construct since all metadata is regular data. Even the user can define data types and add instances of them inside data from third-party applications. The operating system has access to all data by general means. The user can run queries such as:

```
select ~album (having ~guitarist (eq #Eric Clapton));
select ~picture (having ~person (eq #mom));
```

Characteristics of Text-Oriented Software

Let us now conclude by reviewing the characteristics of a text-oriented software system and comparing them to the characteristics of a Unix system that we considered at first.

The key role in text-oriented software is played by text units.

There is a central text repository that provides a unified name space for everything. Queries can be done on everything —from data to procedures, from user customizations to applications and system software— by generic means.

All data and software is represented by text units. Thus, any text editor can update all of them. A text editor can be a programmatic or a visual one, the latter can represent text units linguistically or graphically. Also configuration parameters for software components are set through text units, so that a text editor can control all aspects of the functionality.

Procedures are text transformations, and commands are filters that receive a text unit as input and generate another text unit as output.

The system builds an interactive environment that provides uniform access to all functionality and extensibility by linguistic means. Several user interfaces are supported, from linguistic (shell) to graphical ones.

To use the system, a user builds a command and instructs the system to execute it. One can already guess that the user experience will be similar to Unix. Yet the system will be much smoother and more well-integrated, and will provide more powerful utility.

It will require hard work to get to such systems. But when we've gotten there we will look back to current systems, which we will then experience as clumsy, rough, headless, and be filled with wonder as to how we managed for so long in such an uncivilized manner.

—-oOo—-