Logan Montgomery

CSC362

Prof. McCartin-Lim

4/1/2021

The Unix Philosophy Reflection

Systems that utilize the 17 principles of design:

1. Rule of Modularity: Write simple parts connected by clean interfaces.

Mario, Design of Mario is an ‘object’ based system such that objects displayed on the screen have independent functions. This is an advantage because each object can be debugged individually without effect other objects.

1. Rule of Clarity: Clarity is better than cleverness.

Python enforces this well because much of the code looks very sudo code like it is typically clear what is happening.

1. Rule of Composition: Design programs to be connected to other programs.

ML forces this rule its design. Each function in ML takes only one input and gives one output. Function composition is hoe ML gets its power. This is nice because most functions tend to be similar and can be composed inside most other functions.

1. Rule of Separation: Separate policy from mechanism; separate interfaces from engines.

Any OS that uses a hardware extraction layer. The hardware and software both talk to this intermediate layer to make the correct connection between the software and hardware. This allows multiple different software to run on the same hardware.

1. Rule of Simplicity: Design for simplicity; add complexity only where you must.

Most electrical systems in homes strive to have simple designs. This makes it less likely to cause issues such as fires and makes a safer environment when the overall system is simple

1. Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.

This is similar to simple is better, another system that uses this a theatrical production, not a Hollywood movie, but a Broadway show. They tend to look big and dramatic but behind the curtain its mostly simple systems that build up the entire show. This keeps the cost of production down.

1. Rule of Transparency: Design for visibility to make inspection and debugging easier.

This rule is seen in most vehicles. They are discoverable in that we can determine what all the switches and pedals do, and the dashboard allows for driver to see internal status such as temperature, rpm, fuel level, etc. These are important feedback for the drivers safety and convivence.

1. Rule of Robustness: Robustness is the child of transparency and simplicity.

Chat bots are robust under Unix’s definition. That is, most chat bots will operate with obscured and unexpected inputs. Well, maybe not csc 111 chat bots, but certainly the online chat bots abuse the principle of being robust to fake being human.

1. Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.

C does this, “he C language's facility at manipulating pointers, in particular, has encouraged the use of dynamically-modified reference structures at all levels of coding from the kernel upward. Simple pointer chases in such structures frequently do duties that implementations in other languages would instead have to embody in more elaborate procedures.”

1. Rule of Least Surprise: In interface design, always do the least surprising thing.

Qwerty keyboard. Every keyboard interface operates this way. They use pre existing knowledge that we all have and implement it into a new device. This is advantageous because not only is it easy to switch to a new device, they sell more device when they meet these standards.

1. Rule of Silence: When a program has nothing surprising to say, it should say nothing.

This idea is used in many classrooms, both by students and teachers. Teachers should only say what is important for the lesson so not confuse students with irrelevant information. Students also try not to ask questions that do not pertain to course material. All of these makes it easy to pick out the important information form a lecture.

1. Rule of Repair: When you must fail, fail noisily and as soon as possible.

Many audio/video systems operate this way. They accept many types of input, HDMI, VGA, Optical, RCA, etc, and output the same information for each type, some auditory output or video display. This is advantageous because people can use multiple different types of systems with the same TV set or speaker.

1. Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.

Python does this well. A beginner can pick up the language fast and an expert can write in the language with ease because of its similarities to sudo code. It is a high level program designed to be efficient for writing.

1. Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.

Again, Python does this well with smart compiler designs that allow code to written high level and complied down to machine code. This means more of the work is left up to the computer and less human errors will be present in the code written.

1. Rule of Optimization: Prototype before polishing. Get it working before you optimize it.

The design of many manufacturing facilities follows this rule. When creating an assembly line or manufacturing process you want to make it work first. Once the line I moving and you’re bringing in money you start to invest in efficiency.

1. Rule of Diversity: Distrust all claims for “one true way”.

Many businesses try to incorporate this rule with diversity of people. They want people from many different backgrounds to come wok for the so there is always new insight on their processes. This an increasingly popular among business today.

1. Rule of Extensibility: Design for the future, because it will be here sooner than you think.

Hardware components have be designed this way to ensure they can be used for a long time. One example might be a mother board that supports many slots for ram so It can be upgraded in the future.

Systems that don’t utilize the 17 principles of design:

1. Rule of Modularity: Write simple parts connected by clean interfaces.

WRK, Was not simple parts at all. WRK would be better because when bugs arise they could be fixed.

1. Rule of Clarity: Clarity is better than cleverness.

Java does not do this. Most java code is confusing. It might be clever to force all programs to be object oriented but it’s not always clear what’s happing. If java was clearer, it would be easier to write and maintain.

1. Rule of Composition: Design programs to be connected to other programs.

WRK does not do this. Many functions in WRK output very different data types. This makes it difficult to utilize functions within other parts of the code.

1. Rule of Separation: Separate policy from mechanism; separate interfaces from engines.

A system that does not do this is…

1. Rule of Simplicity: Design for simplicity; add complexity only where you must.

IBM did not have this in mind when they paid their programmers by lines of code. This will inherently breed bad and complex code often with redundant lines. This makes code much longer and having a simple philosophy could improve the code greatly.

1. Rule of Parsimony: Write a big program only when it is clear by demonstration that nothing else will do.

Like I mentioned up top, Hollywood productions are notorious for being over the top, expensive and complex. Typically, the most complicated way to do something is very expensive so we see the budgets for some modern films into multiple millions of dollars.

1. Rule of Transparency: Design for visibility to make inspection and debugging easier.

This may be a really dumb example but it really applicable. Doors. Doors are notoriously non transparent. Bad door designs are everywhere, we’ve all tried to pull a door that needed pushed or the other way around. The are hard to look at and determine whoch way to operate unless they are transparent which man aren’t. Look up Norman doors if you don’t believe this is a huge issue.

1. Rule of Robustness: Robustness is the child of transparency and simplicity.

Unix says robust is when the system operates under un expected conditions. Under this definition, many industrial machines are not robust. Many machines have an over abundance of safety switches which will simply terminate any process if there is some unexpected condition, such as someone’s hand in the way. For this, these systems might actually be worse if Unix considered them “robust”

1. Rule of Representation: Fold knowledge into data so program logic can be stupid and robust.

Program that uses complex code and simple data structure…

1. Rule of Least Surprise: In interface design, always do the least surprising thing.

The mac, “command” key does not follow this. Sometimes it acts like the windows “ctrl” key and sometimes it doesn’t, there is also a “control” key on a mac keyboard. This is a surprising interface because it does not follow the standards that we already know. If mac would have used the least surprise rule more user would feel that mac was compatible with their needs.

1. Rule of Silence: When a program has nothing surprising to say, it should say nothing.

This is not the case with windows. Windows will constantly ask the user for updates and the updates will do absolutely nothing. I could consider this something the system says but does nothing interesting. Even Microsoft programs on mac have this trait, I hate it. Programs would be far less annoying If they followed this Unix philosophy.

1. Rule of Repair: When you must fail, fail noisily and as soon as possible.

As the paper suggest HTML does not follow this because it allows many inputs but does not interpret them well. HTML could be better if many inputs could be used but really they should just be more specific with what they want.

1. Rule of Economy: Programmer time is expensive; conserve it in preference to machine time.

C does not do this, As the paper suggest C is a lower level language that requires more of a programmers time to complete a good program. C however should probably not adapt to this standard because it is useful in other ways being a low level program, making it a high level program easier to write will retract from these values.

1. Rule of Generation: Avoid hand-hacking; write programs to write programs when you can.

Writing ASM code inside of C is a bad example of this, You should allow the complier to create all of the assembly code needed for the program. WRK uses inline ASM.

1. Rule of Optimization: Prototype before polishing. Get it working before you optimize it.

Many regulations in place today make this rule hard to follow when designing a new engine for a car. The fuel efficiency of a vehicle is often the first priority in design, making for complicated fuel management and ridiculously hard to maintain vehicles.

1. Rule of Diversity: Distrust all claims for “one true way”.

Form what I saw, WRK does not do this well. Most code was in C language or ASM and I’m not sure of nay place where another language was used.

1. Rule of Extensibility: Design for the future, because it will be here sooner than you think.

A system that doesn’t do this is poorly designed hardware. Many laptops are not easily upgradable or repairable. Things like ram, processors, even hard-drives, are often soldered onto boards making them difficult or impossible to make future changes to.

If I was to design an OS I would take into consideration all of these ideals but in particular I think modularity, simplicity, and least surprise of the upmost importance. Of course things like robustness and repairability are important but if I was to chose, modularity, simplicity, and least surprise would rules to live by. Modularity is particularly important to ensure that functions can be used any many places and by many processes. It helps ensue that al the code can talk to each other with ease. Simplicity is just good design. It don’t matter if its code or otherwise. Simple is always better, especially in a complex system such as an OS. The rule of least surprise seems obvious but clearly there are examples of OS that does not do this, \*cough\* Windows \*cough\* \*cough\* It is truly frustrating when you’re working on a project and an OS has a “read file” system call that isn’t even used