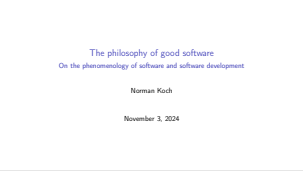
The philosophy of good software

On the phenomenology of software and software development

Norman Koch

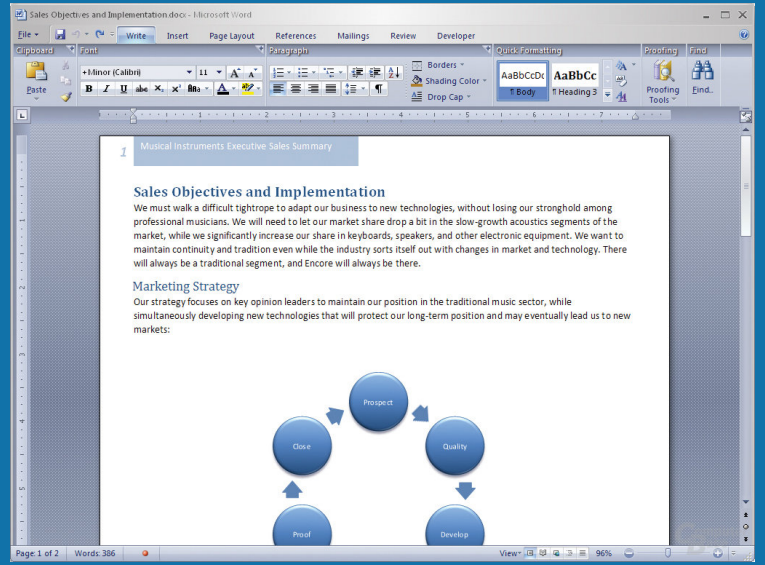
November 3, 2024



Outline

Outline

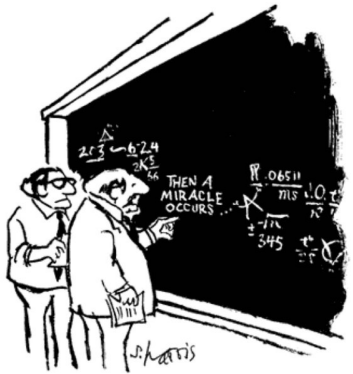
Why care?
Just follow the truth table!
What is »Phenomenology«?
Summary



- Imagine you're working on an important document and suddenly...
- Nothing is working anymore, your document probably lost.
- You don't perceive this as opportunity to learn more about Computers, how they work and why they fail
- It's not an invitation to get more familiar with them. (Even though it absolutely is.)
- It's the task that's important, the computer is just a way to solve a task, it should not be the cause of other problems that maybe you have no idea how to deal with

Let's always remember this on our way to develop better software.
Good software is that which does **not** annoy like that. And only good Software gets sold and used. This is why you should care.

Let's always remember this on our way to develop better software.
Good software is that which does **not** annoy like that. And only good Software gets sold and used. This is why you should care.



"I think you should be more explicit here in step two."

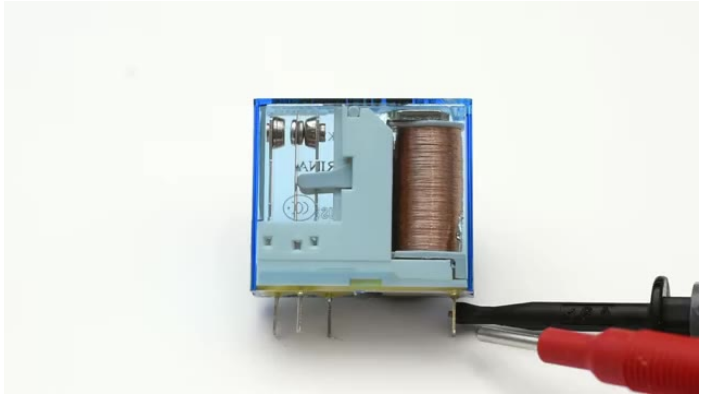
Harris, *What's so Funny about Science?*

Harris, *What's so Funny about Science?*

- Before we can develop really good software, we need to demystify the computer.
- For some people computers are entirely like magic. You put something in – then, magic happens – and you get something out
- Like there's a magic elf inside that solves these problems for you. Even for programmers.
- For some, like me, they're not *entirely* magic anymore. I'm far from knowing all the details, but I think I got the gist of it right, and so I want to explain to you what a computer is, in philosophical terms, without the assumption of magic.

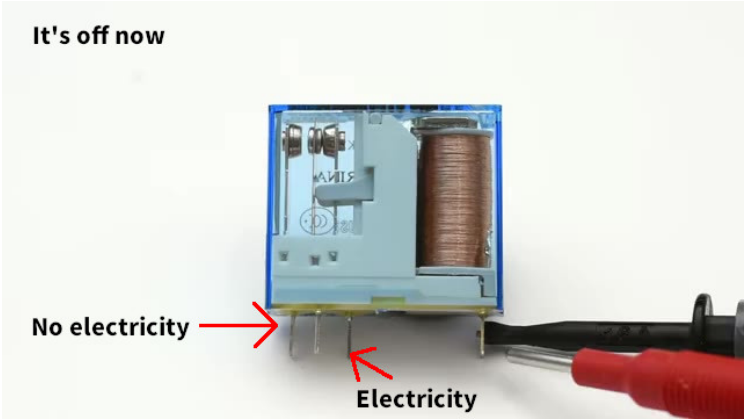
- I assume we have relays and switches.
- A switch interrupts a flowing current when it's in it's OFF position
- A relay has two inputs, a general-power input and a control input.
- It's function is to output power, once the control switch is on and not output power when it's off.

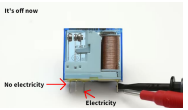
- I assume we have relays and switches.
- A switch interrupts a flowing current when it's in it's OFF position
- A relay has two inputs, a general-power input and a control input.
- It's function is to output power, once the control switch is on and not output power when it's off.

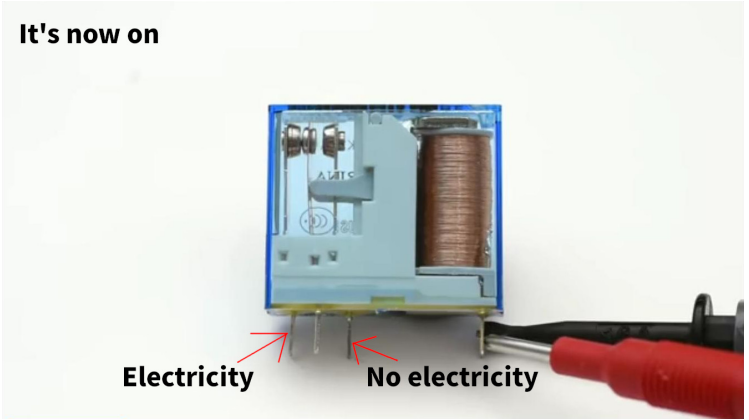


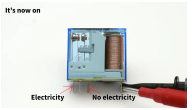
File:Relais-Finder-12A.webm — Wikimedia Commons, the free media repository

File:Relais-Finder-12A.webm — Wikimedia Commons, the free media repository


It's off now
No electricity
Electricity


It's off now
No electricity
Electricity


It's now on
Electricity
No electricity


It's now on
Electricity
No electricity

- A function has an input-vector and an output-vector.
- $z = f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} x^2 \\ x + y \\ y^2 \end{bmatrix}$ for example
- Or `AND`: $z = f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = [x \wedge y]$

- A function has an input-vector and an output-vector.
- $z = f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = \begin{bmatrix} x^2 \\ x + y \\ y^2 \end{bmatrix}$ for example
- Or AND: $z = f\left(\begin{bmatrix} x \\ y \end{bmatrix}\right) = [x \wedge y]$

- For this, let's very quickly turn to formal logic and math. I assume you have a basic idea of what that is.
- Let's imagine, our input x and y can only be one or zero, nothing else and keep the coventions for operations (this called a *ring of integers of modulo n*)
- We want to define one of the most basic parts of a CPU, an ALU (algorithmic-logical unit).
- There are many more parts to a CPU, like control flow units, but we are going to ignore those.
- Let's define an `AND`.

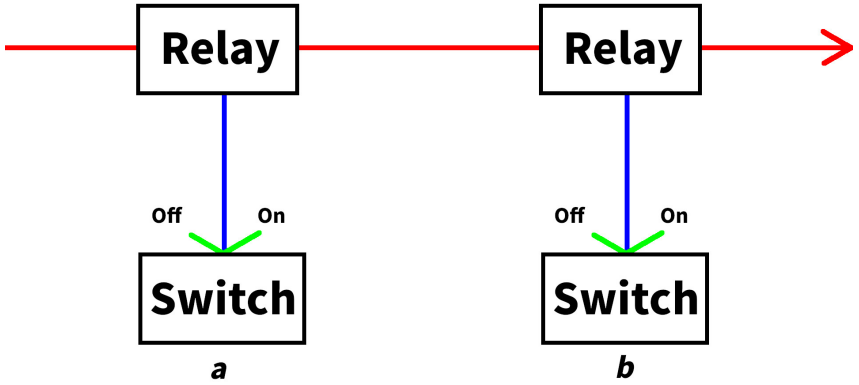
Let's look at the truth-table of `AND`.

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

Let's look at the truth-table of AND.
a | b | a AND b
0 | 0 | 0
0 | 1 | 0
1 | 0 | 0
1 | 1 | 1

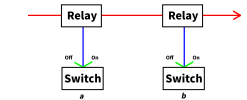
- See it? It's only true when both a and b are true.
- Let's construct this.

a AND *b*:

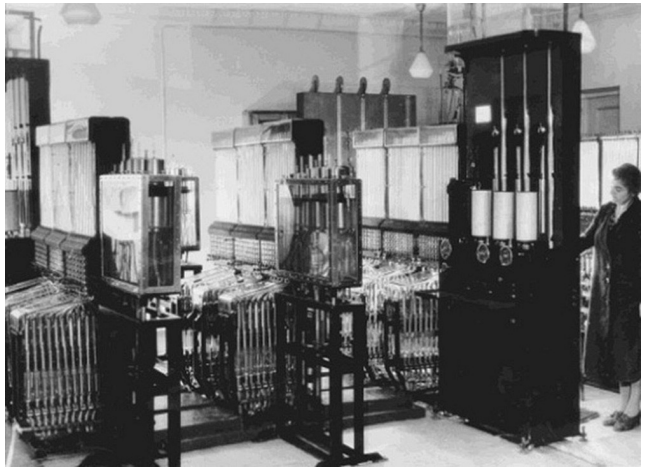


└*a* AND *b*:

a AND *b*:



- See? This can only be ON once both the *a*-switch and the *b*-switch are on. Otherwise, nothing comes out of this circuit.
- Now, this is not only a circuit, but the starting moment of modern computing.
- The idea is to use the natural laws of the universe (like the laws of electricity or flowing water) in such a way that they can represent a truth table
- All modern computing is based on the creation of analogies of internal mental structures in the outside-world
- (I personally believe that this is because logic itself is an analogy of the world, so that we have the ability to manipulate things in our mind, and this ability is what differentiates humans from all other animals. Computers allow us to outsource the simplest form of thinking to a machine.)
- Only in the moment of inputting something and the moment of reading the outputs, the computer is really ‘calculating’. Inbetween, it’s just doing physics.



Patowary, *Vladimir Lukyanov's Water Computer*

Patowary, *Vladimir Lukyanov's Water Computer*

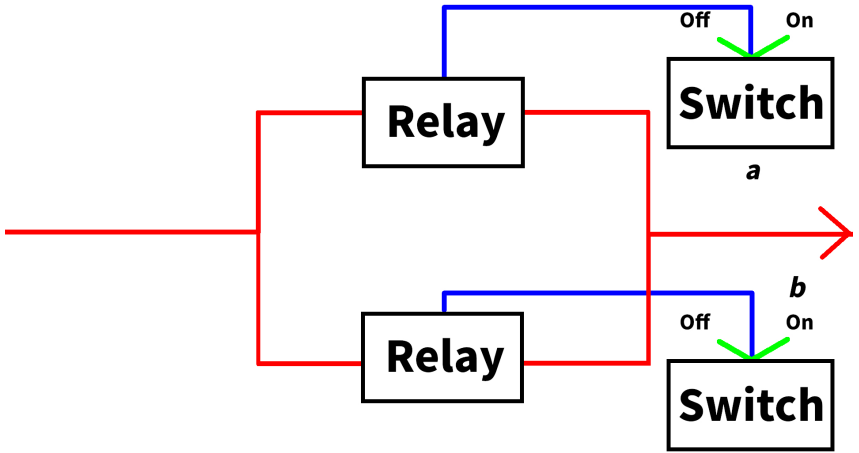
- This analogy-idea of a computer can be transferred to quasi infinite many different ways.
- You can build computers with relays or with transistors. For the logic, it doesn't matter.
- You can even build computers that run on water, because you could choose water instead of electricity.
- This has been done btw.
- You even could build this from vacuum chambers and vacuum valves. With that idea, either there is a good vacuum (1) or there's not (0). You can literally, in principle, compute anything computable with *nothing*. You could run Linux on something that is the most *nothing at all* we can achieve physically, if time and money were really no concern.

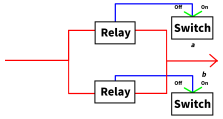
Now let's define OR via it's truth table.

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

Now let's define OR via it's truth table.
a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

a OR *b*:





- This is a logical OR. It is true when at least one input is true.
- At the end, there must be electricity (or water, or vacuum) if one or both switches are on.

- One last basic definition, but this one is very simple. Let's look at the truth table from `NOT`.

a	NOT a
0	1
1	0

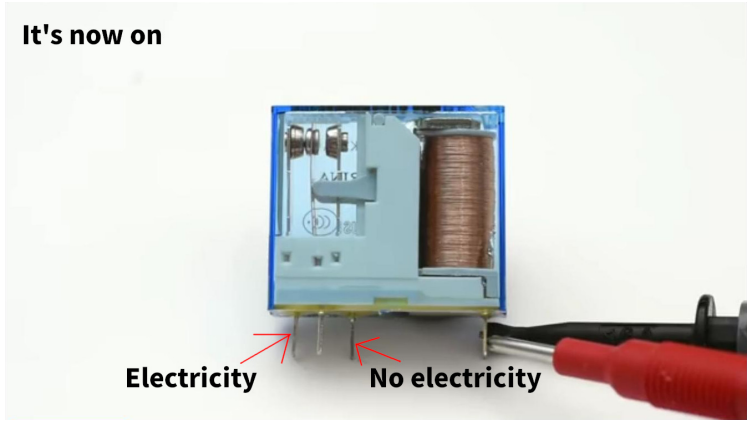
- This can easily be achieved by connecting the cable to the rightmost pin, so that there is no current flowing when the switch is on.

- One last basic definition, but this one is very simple. Let's look at the truth table from NOT.

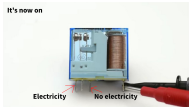
a	NOT a
0	1
1	0

- This can easily be achieved by connecting the cable to the rightmost pin, so that there is no current flowing when the switch is on.

NOT *a*:



NOT *a*:



- See the three pins on the left? The middle one connects to the mains, and the left one is normally closed, and the right one is normally open.
- You can use the right one that is normally open as `NOT`. It only closes once the switch is in the on-position.

- Let's define `XOR`.

- Let's define `XOR`.

- I hope you now see the connection between truth tables and the hardware and could construct something like this.
- Let us now go on even further a bit, but with equations only.
- Let's define `XOR`.

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

- You can see: *it's only on when either (a is on and b off) or (a is off and b is on).*
- (a AND NOT b) *or* (NOT a AND b)
- We have all of these. So we do not need new hardware part designs, but only to merge already existing modules (`AND`, `NOT` and `OR`).

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

- You can see: *it's only on when either (a is on and b off) or (a is off and b is on).*
- (a AND NOT b) *or* (NOT a AND b)
- We have all of these. So we do not need new hardware part designs, but only to merge already existing modules (`AND`, `NOT` and `OR`).

- Imagine putting a lamp at the end of this.
- If we leave 'is on', it is directly the logical notation equivalent to XOR. And this is all we need.
- From this, you can construct `XOR`. You now saw how each of them is realized. You've seen `AND`, `OR` and `NOT`.

- You can also now construct `NAND`, the negated `AND`:

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

- You can also now construct NAND, the negated AND:

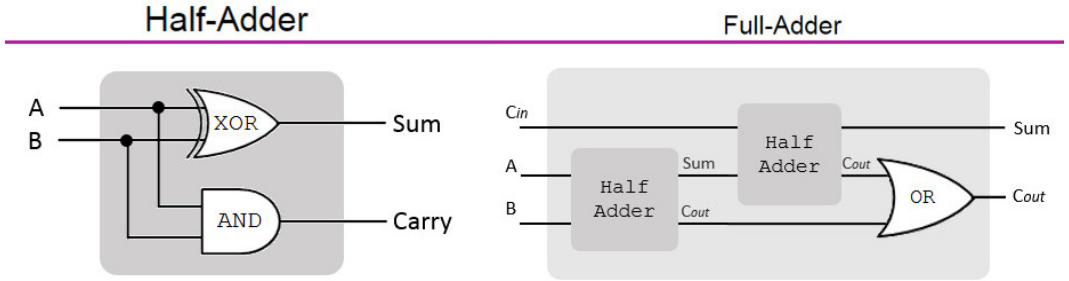
a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

- Or `NOR`, the negated `OR`:

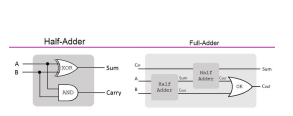
a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

- Or `NOR`, the negated `OR`:

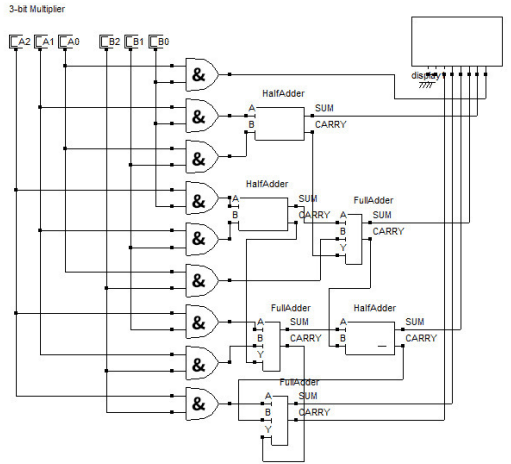
a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0



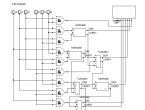
Kerampran, *Binary Additions using Logic Gates*



- From `AND`s, `OR`s, `NOT`s and `XOR`s, you can build adders, subtractors, multipliers and so on. Each one cares less about the direct implementation of the lower layers. It gets more abstract.
- The final layer should be the most abstract: the user-level of the program. You have good software when the user does not need to worry about *any* of the underlying details.
- And you can get more and more abstract (that is, removed from concrete hardware). Once you have a module you know acts like an Adder, just label it Adder and don't care about the exact underlying structure. Do this in all of your programs, so that the end-user does not need to care about implementation details in any way.



Hussaini, *Multiplier – Designing of 2-bit and 3-bit binary multiplier circuits*



Hussaini, *Multiplier – Designing of 2-bit and 3-bit binary multiplier circuits*

- For a general outline, we just need the `JMP`-instruction now.
- `JMP` is able to 'jump' from one part of the code to another, similiar to `goto` in the basic idea
- Most CPUs define multiple jump-functions like `JE` (jump if equal), `JNE` (jump if not equal), `JG` (jump if greater), ...
- With jump-instructions, you can do branches, very similiar to this Pseudo-Code:

- For a general outline, we just need the JMP-instruction now.
- JMP is able to 'jump' from one part of the code to another, similiar to goto in the basic idea
- Most CPUs define multiple jump-functions like JE (jump if equal), JNE (jump if not equal), JG (jump if greater), ...
- With jump-instructions, you can do branches, very similiar to this Pseudo-Code:

```c
if(a == 5) { // jump if a == 5
        goto A_EQUAL_FIVE;
} else { // jump if a != 5
        goto A_NOT_EQUAL_FIVE;
}
return;
A_EQUAL_FIVE:
printf("a==5");
return;
A_NOT_EQUAL_FIVE:
printf("a!=5");
return;
```


if(a == 5) { // jump if a == 5
goto A_EQUAL_FIVE;
} else { // jump if a != 5
goto A_NOT_EQUAL_FIVE;
}
return;
A_EQUAL_FIVE:
printf("a==5");
return;
A_NOT_EQUAL_FIVE:
printf("a!=5");
return;

- When you have adders, you can build multipliers of them (as, with the integers at least, multiplication is just repeated addition).
- When you have multipliers and branch-statements like `JE`, you can use them to create a faculty function like that:

$$n! = \begin{cases} 1 & \text{if } n == 1 \\ (n-1)! & \text{else} \end{cases}$$

- When you have adders, you can build multipliers of them (as, with the integers at least, multiplication is just repeated addition).
- When you have multipliers and branch-statements like JE, you can use them to create a faculty function like that:

$$n! = \begin{cases} 1 & \text{if } n == 1 \\ (n-1)! & \text{else} \end{cases}$$

Assuming you also have a divison circuit, which we will also skip here, you can then build:

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} \mp \cdots$$

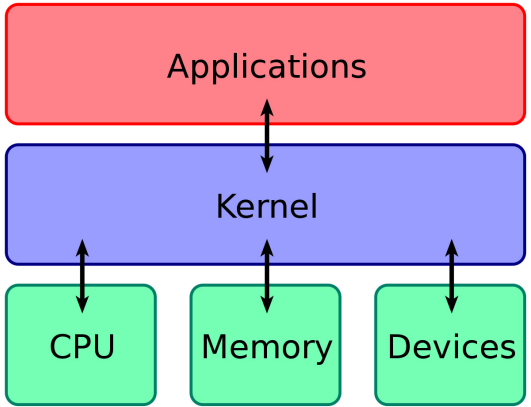
And with that work on even more and more complex structures, not caring about the detail-implementation anymore.

Assuming you also have a divison circuit, which we will also skip here, you can then build:

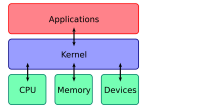
$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} = \frac{x^0}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} \mp \cdots$$

And with that work on even more and more complex structures, not caring about the detail-implementation anymore.

- This is what libraries do for you.
- When you do $\sin(x)$ in C, it gets breaken down to the taylor series shown
- This then gets broken down to a lot of `AND`s, `XOR`s and `NOT`s.
- Programs as I think most programmers understand them are usually only 'glueing together' libraries, things that abstract away from the hardware that you don't have to think about the logical implementation.
- When it's there, the sin can be used to calculate all kinds of things. Every 3D-game you ever played used these as libraries for the graphics for example. And the developer did not need to re-invent the wheel.

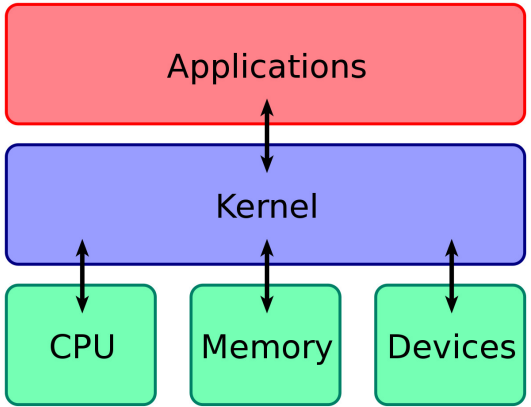


Commons, *File:Kernel Layout.svg — Wikimedia Commons, the free media repository*

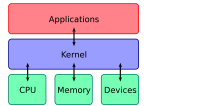


Commons, *File:Kernel Layout.svg — Wikimedia Commons, the free media repository*

- The quest for more and more abstract ways of programming leads us to operating systems.
- An OS is a piece of software that abstracts away the concrete hardware.
- The *kernel* is what communicates with the hardware, and your software communicates with the kernel.
- The kernel allows you to execute certain commands, abstracted away from the real hardware, so you don't need to care about a specific implementation of `AND`s and `XOR`s, but only about doing basic level mathematics for example.



Commons, *File:Kernel Layout.svg — Wikimedia Commons, the free media repository*



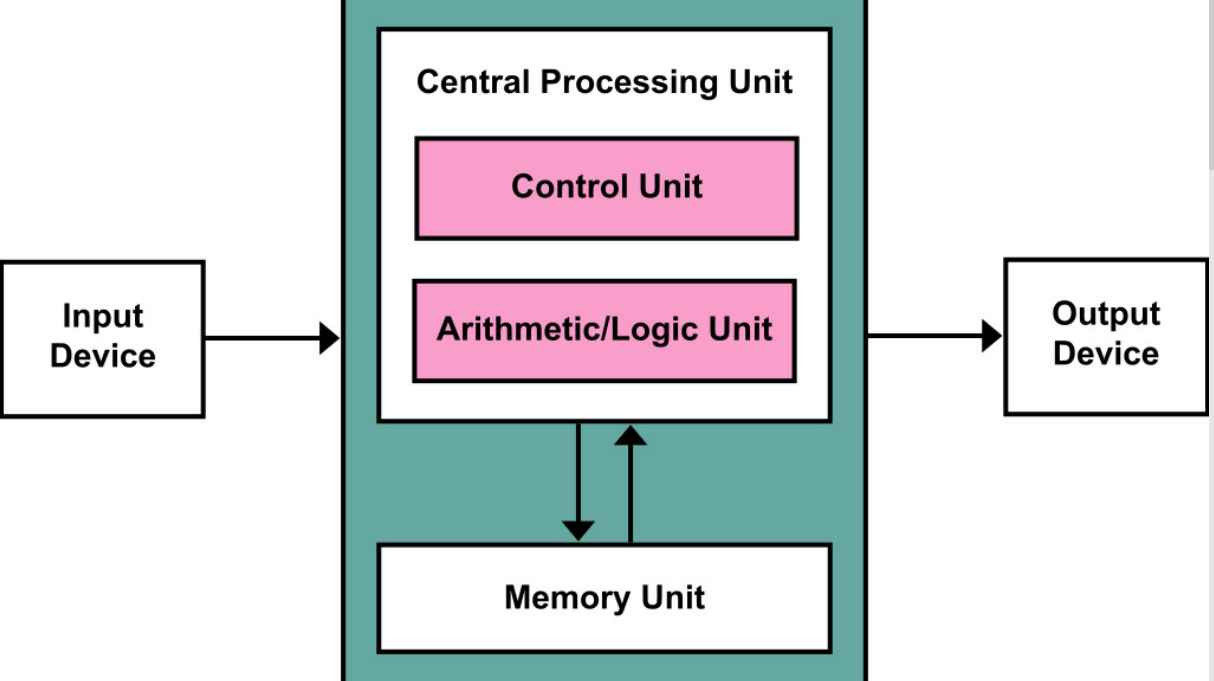
Commons, *File:Kernel Layout.svg — Wikimedia Commons, the free media repository*

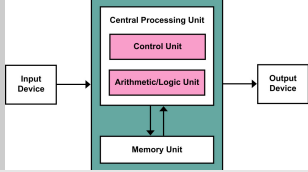
- Having a kernel has the advantage that you don't need to care about communication with the hardware, only with the kernel. The kernel communicates with the specific hardware for you.
- The OS also allows you to communicate with other hardware, like printers, over a generalized interface. The OS needs a driver (a piece of software that tells the kernel how to use certain pieces of hardware), and then you can use the OS's features with that hardware, without caring about the intricacies of the printer.

- This is also what higher-level-languages do.
- The idea of a high level programming language is that, for programming them, you don't need to care about implementation details.
- They abstract things away from you. And I guess it's easier to find a python developer than a assembly-expert because of exactly that.
- Coding in assembly is much harder because you need the ability for higher mental workload.
- Not only the problem you really care to solve, but also additional problems (implementation details)

- This is also what higher-level-languages do.
- The idea of a high level programming language is that, for programming them, you don't need to care about implementation details.
- They abstract things away from you. And I guess it's easier to find a python developer than a assembly-expert because of exactly that.
- Coding in assembly is much harder because you need the ability for higher mental workload.
- Not only the problem you really care to solve, but also additional problems (implementation details)

- Please always remember that all of this would not have been possible without philosophers like Aristotle, who first had the idea of formalizing logic, or George Bool, who first researched 'boolean arithmetic', or Descartes, who first researched function theory.
- Most of those discoveries were, at their time, regarded as 'useless'
- See '*A Mathematician's Apology*' by G. H. Hardy (1940), in which he claims that number theory, is one of the most beautiful mathematical areas, despite it not »*being able to be used for anything*«
- He was right and wrong at the same time. It is the most beautiful mathematical area, but today also one of the most useful ones. Without developments in number theory, the internet would not have been possible as it is today (no cryptography would be possible without advancements in math made some houndreds or even thousands of years ago that, at their time, seemed useless to most people).
- So remember, sometimes 'the most useless' things may turn out to be the most useful things to later generations.





- What I've skipped:
- My description of what a computer is is far from complete here. For it to run as you are now used to, you need memory, for example, and you need way more transistors than I could possibly draw. Also, you need a control unit for your CPU. And so on.
- But this should give you just the right gist of thinking about the central part of your computer that is usually more mystified than clear to most people.
- If you want to learn more about this, search for *Von-Neumann-Architecture*. This here was just intended to demystify the very basics.

- What does this practically mean?
- You must try to `catch` any errors you can imagine. Assume everything can – and will – fail all the time in every conceivable way. And also in all ways that you did not consider.
- Did you, for example, know that `fork()` can fail and that, if not properly treated, has disastrious consequences?[1]
- Also assume everything that can be entered by a user is garbage. They'll add numbers in fields designated for letters, they enter 0 in a field through which something is divided and they'll try to `chmod +x` and run any `jpeg`-file they own and run them with sudo.

[1] Rachelbytheway. *fork() can fail: this is important*. URL: https://rachelbythebay.com/w/2014/08/19/fork/.

- What does this practically mean?
- You must try to catch any errors you can imagine. Assume everything can – and will – fail all the time in every conceivable way. And also in all ways that you did not consider.
- Did you, for example, know that fork() can fail and that, if not properly treated, has disastrious consequences?
- Also assume everything that can be entered by a user is garbage. They'll add numbers in fields designated for letters, they enter 0 in a field through which something is divided and they'll try to chmod +x and run any jpeg-file they own and run them with sudo.

Rachelbytheway. *fork() can fail: this is important*. URL: https://rachelbythebay.com/w/2014/08/19/fork/.

- Also really make sure you trust your subcomponents. Proving every algorithm is not do-able in practice, but write as many unit tests as you can. And test the interaction between modules, too. Best auto-test all of the programs features.
- Write each positive and negative tests, so that you can test your error routines. If you find bugs, write a test that matches them first, and then fix them. Then you can test for the presence and absense of that bug.
- Monkey-Test your script. For webpages, I recommend *gremlin.js*. You can run a simple piece of code and it will click randomly and enter anything into any field and press any key randomly, with as many workers in parallel as you wish.
  - For X11 applications, you may look into `xdotool` (or `ydotool` for wayland).

- Also really make sure you trust your subcomponents. Proving every algorithm is not do-able in practice, but write as many unit tests as you can. And test the interaction between modules, too. Best auto-test all of the programs features.
- Write each positive and negative tests, so that you can test your error routines. If you find bugs, write a test that matches them first, and then fix them. Then you can test for the presence and absense of that bug.
- Monkey-Test your script. For webpages, I recommend *gremlin.js*. You can run a simple piece of code and it will click randomly and enter anything into any field and press any key randomly, with as many workers in parallel as you wish.
  - For X11 applications, you may look into `xdotool` (or `ydotool` for wayland).

- You cannot trust anything from the outside. Don't assume a User will only use `ASCII` in his folder names, but assume strange `UTF-8` symbols from a russian encoded website from a buggy browser that modifies random bits. It must either work, if you really find no way of making it work, display an exact error message on what's wrong and what needs to be done about this. (When writing GUI-Applications, make it easily copyable so that it can be googled).
- Like in real life, you cannot stop things from going wrong. But you can stop them from going horribly wrong. You can catch most of the 'wrongness' if you clean up after yourself properly and don't rely on other people to do your work

- You cannot trust anything from the outside. Don't assume a User will only use ASCII in his folder names, but assume strange UTF-8 symbols from a russian encoded website from a buggy browser that modifies random bits. It must either work, if you really find no way of making it work, display an exact error message on what's wrong and what needs to be done about this. (When writing GUI-Applications, make it easily copyable so that it can be googled).
- Like in real life, you cannot stop things from going wrong. But you can stop them from going horribly wrong. You can catch most of the 'wrongness' if you clean up after yourself properly and don't rely on other people to do your work

- Heidegger was the first philosopher to criticize the european philosophy at it's fundamental level.
- He said, from Plato on, philosophers have lost track of 'real life', focussing on abstractions instead.
- Using abstractions is easier of course, but an abstraction can never replace real life.
- He called it 'Seinsvergessenheit', the forgetting of being itself.

- You can think about riding a bike, or ride a bike. They're not the same at all.
- This is called ontic-ontological difference. Ontic refers to 'what we really do', ontological is 'thinking about what we really do'.
- The task of philosophy is, according to Heidegger, to get the the focus to the real life instead of just thoughts about real life.

- This is what we have to keep in mind for good software, too.
- Move away from what you think users want and are to what users really want and are.
- This is not easy, since most of the time they don't know it themselves.
- But it can be achieved by observation of yourself, since you are also 'just' a user of your own software.
- And it can be achieved by outside user's with different mindsets and goals via communication.
- Schmitz, a successor of Heidegger, has really started this focus on real life, in my opinion.

- This is Hermann Schmitz.
- He's a comtemporary german philosopher who invented 'the new Phenomenology'
- Sadly, he passed away on the 5. of may 2021 at the age of 92.
- His ideas in short terms were:
  - We are not merely our rationality, nor are we merely our body.
  - Our rationality is that which thinks most often in words inside our head.
  - Our body is that which a doctor palpates when he examines us.
  - What's missing is our *subjective body*. If a loved one touched you exactly the same as a doctor may touch you, it may feel very different.
  - The subjective body is our experience of our body *and* our mind.

- Do you know phantom pain?
- If someone loses an arm due to an accident, it would be logical to think that he does not feel it anymore.
- But this is not the case. People who lost limbs often report still feeling the pain in them.
- This is because though they lost a part of their physical body, it is still part of their *subjective body*.
- In this *subjective body* we experience all our emotions and atmospheres.
- All of live is basically just a row of following situations in certain spaces, submerged in atmospheres.
- This is important to notice. You cannot not be in a situation. You cannot not be in an atmosphere. They sorround you all the time.

- Atmospheres and situations, not physical matter, primarily constitute reality as we experience it
- The new idea of Phenomenology is to treat experiences as real. The pain you may experience is as real as the chair your sitting on
- They are in different categories, since you have access to that chair, but not my experiences, but they are nonetheless the realest reality I can get
- If you are in severe pain, the pain is much more real to you than a mathematical proof
- Good programs create a good atmosphere. Programs that create bad atmospheres (by, like, crashing, or acting unexpected) are bad programs.

- !!! Add creepy music here !!!
- Imagine walking through this abandoned insane asylum in the night without a flashlight, hearing weird sounds
- Do you feel that constricting feeling in your chest-area? This is your *subjective body*s reaction to that creepy, weird area.
- The german Word ‘Angst’ is etymologically derived from the same word as ‘Enge’, which means ‘narrowness’, because it’s connected to this constricted feeling.

- Or have you ever felt like this? Liberated, euphoric, happy to be alive?
- The feeling in your chest-area might be liberating and widening.
- According to Schmitz, these are the most general ways of subjective body states: between absolutely constricted, and absolutely open
- You need both in life, like the expansion of the Leib when breathing in in swelling of the fealing chest-area, that would be unbearable if it wouldn't be exhausted just right after, resulting in in wave of experiences of widening and narrowing Leib
- Usual situations are a complex mixture of states between those extremes in different what Schmitz calls 'Leibliche Region', like the lost arm with phantom pain, "that are in the area, but not neccessarily in the borders of the body".
- Whether you feel fear, or you are happy or liberated, it is always a feeling that affects the Leib

- These are the most basic extremes of life. This is the best that 4 billion years of evolution could produce to tell us about our sourroundings.
- We should start to use it properly!
- That is, have our inner focus properly adjusted to experiences.
- If you use your software, and something annoys you, which you can fix, *fix it*! (This is a generally good tip for life)
- But you have to really use the software. And see how other people use your software. They'll often don't see things that seem 'obvious' to you because your so involved with your code.

- They also see things that are obvious to them, but not to you, since you are not thinking like them, because you are not them.
- Heidegger calls this “Jemeinigkeit”, the *Dasein*, i. e. *you* as a human being, is always experienced from the subjective perspective that is always *mine* (from the person observing)

RTFM

ThankYou

- In theory, users have an infallible memory system. And they read manuals before using the program. They know exactly what they want and how to achieve it the most efficient way possible.
- But in reality they are never like that.
- Do you read a 400 page manual before using a ‘simple’ program? I do not. Nobody wants to read a manual to use a webbrowser. It’s just a tool that needs to do it’s job. And of course real humans always forget and barely ever know what they want exactly.
- It’s not like when you play music on Youtube, you want to think about how the https-protocol is implemented. You only care about the music.

RTFM

ThankYou

- Thinking is a very complex task and requires a lot of effort that the people using software would rather like to spend on the problem they try to solve *with* the computer
- I believe we developed consciousness evolutionary to have the ability to focus on problems that we cannot solve by intuition alone.
- Remember when first learning to drive a bike or a car? You had to do everything very consciously, until it went 'down in your nervous system' so that you do the things correctly without thinking about them. When consciousness fulfils it's duty of learning, we can disable it and drive on auto-mode mostly.
- The task of consciousnes is, as Zen buddhism puts it, to get rid of consciousness.
- This is what he have to achieve with our software, so that people will use it without having a learning curve. If your users always use the program wrongly, they think different about the problems than you, and your task is to bridge that gap by making it work the way people think.

RTFM

ThankYou

- Real humans have a very limited ability for mental workload. They can only keep some very few things at the same time in their head, and usually the computer is one of those that they have to have in their head, but don't want it there, because they don't have the goal of using the navigation software, but of reaching the goal *with the navigation software*.
- They want a result on the level of analysis they look at right now
- If you drive a car and it breaks down, you have to change your level of analysis. From *this is an object that gets me from A to B*, you must suddenly switch to *this object has an engine with like a billion parts and complex circuits and all that stuff, what do now?*.
- A good car is one where you don't have to switch the level of analysis. And this is also the trademark of great software.

- Hegel wrote an article about abstractness and concreteness (Hegel, *Wer denkt abstrakt*). He said, usually people think the abstract thing is the hard one, like abstract mathematics, and the concrete is the easy one. He argues that often it's the opposite.
- For example, checking my average school marks vs. the ones' of another one is easy. But looking at whether the person is right in that job, that's hard.
- Abstract means, we exclude. It is etymologically derived from abstrahere in latin, which means to 'cut away'. Abstract means we can concentrate on only one or at least very few factors.
- Abstraction is great, because then we can see parts of the future and develop towards them, even though they are not here yet. This gives us the vision to do things. But it comes with the price that we all-to-often forget the reality, fall for Heideggers' 'Seinsvergessenheit', and confuse our image of the world with the real world.

- In software, we want exactly t h i s abstraction.
- Good software cuts away everything from you that you don't care about while solving the problem

- Do you know the feeling of flow? When you do something and everything works as expected and helps you to fulfil your goal?
- Sometimes, when driving through tunnels, you see signs that the radio will not work.
- You want to drive a car and listen to radio. Of course you know why it has problems. You kind of know how the radio network works.
- But This is an interruption nonetheless. Where the flow is interrupted.
- NEVER EVER BREAK THE FLOW
- Flow-breakers are needing to think about the implementation of something that you just want to use, like the navigation software, or they even outright stop working, like the radio.

- Every time something breaks the flow, fix it.
- We cannot expect all our users (and ourselves) to become full “zen-monk-flowy”. So we have to take care about all the situations that break the flow by fixing them.
- Think in user-perspective. The user does want to solve a problem, not first create a bunch of other problems for himself.
- While there are flow-breakers, fix them. Fix the most severe ones first (like not being able to start the software without a bluescreen).
- The second biggest flow-breaker has become the most important show-stopper now, which you also fix. The software is 'done' if neither you nor the end users (or a test group of normal users) find some other flow-breakers.
- (But in reality, it's never done.)

- Eat your own dogfood! Actually use your software. And look at how end-users will use it. They will think about it differently, because they don't have the knowledge about it you do.

- Things like speech recognition does all of this. The user does not have to think about a machine as machine, but can only say what they want
- With the rise of language models like chatGPT, this development is much further driven than ever imagined in scifi-movies already.
- This abstracts away completely even from using the ‘normal’ interface
- This is Google’s trick. You don’t need to care about implementation details at all.
- The web usability tester Steve Krug said about this: ‘Don’t make me think!’
- This is the secret.
- Good software is the software that doesn’t make you think

Good software has these 2 properties:

- it is as abstract as possible,
- it **never** breaks the flow!

Good software has these 2 properties:
- it is as abstract as possible,
- it **never** breaks the flow!

Those are the steps from a phenomenological standpoint that you have to follow to create good software:

- Look what you yourself (and others with other mindsets) *really* do and fix the program until it matches the way people who will use the software think
- Try seeing what 'annoying' means by completing the userinyerface.com task to 'register'. You will learn a lot by doing this.

Those are the steps from a phenomenological standpoint that you have to follow to create good software:
- Look what you yourself (and others with other mindsets) *really* do and fix the program until it matches the way people who will use the software think
- Try seeing what 'annoying' means by completing the userinyerface.com task to 'register'. You will learn a lot by doing this.

[1] Free Stock Footage Archive. *Defect Screen - Glitch Effect Footage Free*. 2019. URL: https://www.youtube.com/watch?v=Qr97yriMLBo.

[2] Wikimedia Commons. *File:BSOD Windows 8.png — Wikimedia Commons, the free media repository*. [Online; accessed 20-March-2021]. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:BSOD_Windows_8.png&oldid=491631615.

[3] Wikimedia Commons. *File:Hermann Schmitz Foto.jpg — Wikimedia Commons, the free media repository*. [Online; accessed 20-March-2021]. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:Hermann_Schmitz_Foto.jpg&oldid=461671795.

[4] Wikimedia Commons. *File:Jumping Giulia Salar de Uyuni Bolivia Luca Galuzzi 2006.jpg — Wikimedia Commons*, [Online; accessed 21-März-2021]. 2021. URL: https://commons.wikimedia.org/w/index.php?title=File:

Those are the steps from a phenomenological standpoint that you have to follow to create good software:

- Look what you yourself (and others with other mindsets) *really* do and fix the program until it matches the way people who will use the software think.
- Try seeing what 'annoying' means by completing the userinyerface.com task to 'register'. You will learn a lot by doing this.

Jumping_Giulia_Salar_de_Uyuni_Bolivia_Luca_Galuzzi_2006.jpg&oldid=530222433.

[5] Wikimedia Commons. *File:Kernel Layout.svg — Wikimedia Commons, the free media repository*. [Online; accessed 24-March-2021]. 2021. URL: \url{https://commons.wikimedia.org/w/index.php?title=File:Kernel_Layout.svg&oldid=538228326} (cit. on pp. 29, 30).

[6] Wikimedia Commons. *File:RTFM (12034282963).jpg — Wikimedia Commons, the free media repository*. [Online; accessed 21-March-2021]. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:RTFM_(12034282963).jpg&oldid=477939492.

[7] Wikimedia Commons. *File:Von Neumann Architecture.svg — Wikimedia Commons, the free media repository*. [Online; accessed 24-March-2021]. 2020. URL: \url{https://commons.wikimedia.org/w/index.php?title=File:Von_Neumann_Architecture.svg&oldid=481743644}.

Those are the steps from a phenomenological standpoint that you have to follow to create good software:

- Look what you yourself (and others with other mindsets) *really* do and fix the program until it matches the way people who will use the software think.
- Try seeing what 'annoying' means by completing the userinyerface.com task to 'register'. You will learn a lot by doing this.

[8] Athreon Corporation. *The Benefits and Challenges of Speech Recognition*. URL: https://www.athreon.com/benefits-and-challenges-of-speech-recognition/.

[9] Simon A. Eugster. *File:Relais-Finder-12A.webm — Wikimedia Commons, the free media repository*. [Online; accessed 20-March-2021]. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:Relais-Finder-12A.webm&oldid=440594179 (cit. on p. 8).

[10] Kristine Hansen. *9 Creepiest Places You Should Probably Never Visit*. 2019. URL: https://www.architecturaldigest.com/story/creepiest-places-you-should-probably-never-visit.

[11] G.H. Hardy. *A Mathematician's Apology*. Cambridge University Press, 1967. ISBN: 9780521095778. URL: https://books.google.de/books?id=3R04AAAAMAAJ (cit. on p. 31).

Those are the steps from a phenomenological standpoint that you have to follow to create good software:

- Look what you yourself (and others with other mindsets) *really* do and fix the program until it matches the way people who will use the software think.
- Try seeing what 'annoying' means by completing the userinyerface.com task to 'register'. You will learn a lot by doing this.

[12] Sidney Harris. *What's so Funny about Science?* 1970. ISBN: 0-913232-39-4 (cit. on p. 6).

[0] Georg Wilhelm Friedrich Hegel. *Wer denkt abstrakt*. 1807. URL: http://www.zeno.org/Philosophie/M/Hegel,+Georg+Wilhelm+Friedrich/Wer+denkt+abstrakt (cit. on p. 49).

[13] Umair Hussaini. *Multiplier – Designing of 2-bit and 3-bit binary multiplier circuits*. 2018. URL: https://technobyte.org/multiplier-2-bit-3-bit-digital/ (cit. on p. 24).

[14] Frank Hüber. *Office 12 mit völlig neuer Benutzeroberfläche*. 2005. URL: https://www.computerbase.de/2005-09/office-12-mit-voellig-neuer-benutzeroberflaeche/.

[15] Philippe Kerampran. *Binary Additions using Logic Gates*. URL: https://www.101computing.net/binary-additions-using-logic-gates/ (cit. on p. 23).

Those are the steps from a phenomenological standpoint that you have to follow to create good software:

- Look what you yourself (and others with other mindsets) *really* do and fix the program until it matches the way people who will use the software think.
- Try seeing what 'annoying' means by completing the userinyerface.com task to 'register'. You will learn a lot by doing this.

[16] Kaushik Patowary. *Vladimir Lukyanov's Water Computer*. URL: https://www.amusingplanet.com/2019/12/vladimir-lukyanovs-water-computer.html (cit. on p. 14).

[17] Rachelbytheway. *fork() can fail: this is important*. URL: https://rachelbythebay.com/w/2014/08/19/fork/ (cit. on p. 33).

[18] Flicker René Spitz. 1959. URL: https://www.prospectmagazine.co.uk/arts-and-books/in-defence-of-heidegger (visited on October 25, 2017).

[19] Christoph Vormweg. *Der Philosoph Georg Wilhelm Friedrich Hegel*. 2020. URL: https://www.deutschlandfunk.de/vor-250-jahren-geboren-der-philosoph-georg-wilhelm-100.html.

Those are the steps from a phenomenological standpoint that you have to follow to create good software:

- Look what you yourself (and others with other mindsets) *really* do and fix the program until it matches the way people who will use the software think.
- Try seeing what 'annoying' means by completing the userinyerface.com task to 'register'. You will learn a lot by doing this.