

# A short History of Software Development

Version 1.0

Date: 11 January 2002

© 2002 by GenerExe

[www.generexe.com](http://www.generexe.com)

---

## 1. Introduction

This document gives a short overview of software development history. The goal is not to be complete from a historical point of view, but to refresh our minds of what we are actual doing.

Many of us grew up with computers and learned to program in a playfull-like manner. There is nothing wrong with that, because “playing” is a very good way to learn. Trying out things, modifying examples and verifying the results in short cycles gives us insight/feeling on how things work. This insight is needed, but only extends to the level on which we operated our trials and does not necessarily gives us insight in what/why/how things work exactly. As a result problems may not be anticipated and our creations may only work reliable in our own well-known environment. How does your software perform on a slower or faster machine, a machine with a small hard-disk, a machine with low bandwidth communciation, etc.

Software development deals with the creation of highly complex and dynamic “systems”. We all agree on that. On that note we can pose the following *rule (1)*:

*The higher the complexity of a system, the higher the risk that it can get into a “faulty” state which was not anticipated! (Be warned...)*

## 2. Hardware

### 2.1 Switches, dials and other mechanics

**It all started with switches being used to represent numbers, combined with a number of math-tricks we all learn in grade-school.** A switch is basically a single entity which can be in several distinct discrete states. A switch can be mechanical or electronical. The only difference is that electricity is MUCH faster and uses less energy to change state.

Imagine a mechanical calculator, which adds two numbers from 0 to 9. The two numbers are set by turning two dials from 0 to the value. Both dials drive the same internal dial, which as a result represents the addition. When the internal result dial turns from a 9 to 0, it in turn drives a second internal dial, representing a higher digit of the result-number. There are more “tricks” to implement multiplication and even division. I.e. a multiplication can be achieved by repeated addition.

Furthermore, multiplying a number by 10 when using decimals, means just stupidly

adding a 0-digit to the front of the number. A guy named “Pascal” built mechanical machines based on these principals.

Other math-tricks are based on boolean logic. I.e. comparing/masking single digits (bits in a binary system) and jumping around in the “program” based on the results.

For this generic adding-machine to work, it does not matter which numerical system you use. Whether it is decimal (0 to 9) or binary (0 to 1). I.e. the switch that turns on your living-room light probably has two states: ON and OFF (or 1 and 0 in a numerical binary representation).

## **2.2 Electronic Switches**

The first modern calculators in the era of electricity used a combination of electricity and mechanics. A relais is a mechanic switch, which is turned ON or OFF by an electric signal. The switch itself could close another electrical circuit driving yet more relais. As an alternative, cards with holes punched in it could close the circuits as well, serving as static external memory to input numbers. The speed of electricity could drive thousands of relais in a couple of seconds. Although faster than purely mechanical driven switches (literally requiring horse-power to drive the connected mechanical dials), the relais-based machines still required significant electricity power to drive all the relais. (The mechanical part of the relais is “moved” by an electrical magnet coil, consuming thousands of times more power

than the equivalent in silicon transistors or integrated circuits). Furthermore, the mechanical movement of the relais is relatively slow, when compared to the speed of pure electricity and the lifetime of mechanical switches is relatively short, due to mechanical friction in the moving parts.

People like Turing and von Neuman provided a mathematical foundation for programmable machines, giving birth to program-memory, data-memory, accumulators and central processing units. Readonly information, like programs or constant variables/tables were programmed either by setting manual switches or –a little later- automatically by using punch-hole cards, to set relais (eletronic switches) for the same purpose. Later on, the relais got replaced by switching tubes, which were used already as amplification devices in analog electronics. And the tubes got replaced by transistors, the first low power, highly integrated silicon switching device.

Information was loaded from magnetic storage devices instead of from punch-cards and fast magnetic read/write memories were developed allowing for increasing amounts of variable information to be stored during program execution.

## **3. Software**

### **3.1 Circuit Design**

The early computers were no more than programmable calculators. Programs –very much like cooking-recipes- described the sequence of simpler calculations required

to calculate some more complex math algorithm.

These machines were programmable, because dedicated devices were just too expensive. To program (re-use) the machine, switches had to be set and/or wires had to be replugged manually. A big part of the “software” design consisted of circuit design and making sure that no fuses were blown. The actual programming (loading of the program) was done by a team of people making the required changes *inside* the system – a room filled with arrays of heat-producing electronics and wiring. Programming software required making semi-hard changes in the hardware. In these days there were no keyboards.

A little later, punch-cards provided a faster input method, and cathode-ray-tubes (CRT, like the one in your television set) were used to present results. Keyboard-equipped machines were used to turn human-readable text into punch-cards (like an early assembler).

On this note I want to pose the **rule (2)**:

*Literally SoftWARE means “Soft”  
HardWARE,  
which stands for pliable electronics.*

### 3.2 Scale

The previous section described how the trend in hardware has been: faster, smaller (hence more elements), lower power, etc. High integration of “switches” in silicon, mass production (lower prices). Followed

by the introduction of keyboards, displays and large long-term storage devices. The basic workings of the machines are the same, but the programming no longer requires physical labour nor the replacement of fuses. Over time program design and loading were fused into relatively short test-iterations, which ended when the program produced the correct results.

On this note I want to pose the **rule (3)**:

*The ease of loading a program, combined with the playful manner in which we learn operating computers, gives us the false feeling of security that creating good programs is easy.*

The increase in hardware-capabilities resulted in an exponential increase of potential software complexity. (The more combinations are possible, the more can and will go wrong – Murphy’s law). However, the development disciplines to deal with increased software complexity did not evolve this fast. Sure we have seen higher languages, functional decomposition, reusable libraries, abstract datatypes and objects. Hardware and operating systems helped out by providing multi-threading (parallel recipes) and (protected) memory management. Etc.

These techniques and mechanisms are both meant to help us better formulate solutions –sometimes for specific kind of problems– as well as helping us to control complexity by enforcing certain rules, about how to group chunks of software. Enforcing maintainable software structure, without reducing expressiveness. At the lowest

coding-level we are still doing the same as in the early days: writing recipes.

On this note I want to pose the **rule (5)**:

*One of the biggest challenges of Software Engineering right now is how to control complexity. Efficiently and clearly expressing a solution, which remains maintainable during (and beyond) the software's expected lifetime.*

#### 4. Conclusions

The big blue screen of death, which may occur when your PC-program crashes is like the blowing fuse of 60 years ago. Are interpreters like the ones of Java and Visual Basic really doing us a favour by turning a programming error/bug into a harmless friendly message box? Off course they do, but does it help us to make better programs or just stimulate the playful trial-and-error attitude, which seems to dominate software development for a long time? Imagine the big blue screen of death in an airplane's control-computers or your VCR.

Everybody can program. Computers are affordable to many people and a lot of environments can cope with the occasional program-crash. Programming skills are increasingly high in demand and technology changes so fast that it is hard to keep up with the latest of the latest.

On this note I want to pose the **rule (6)**:

*Most software development is not really engineering and does not have to be.*

However, sometimes *software development* NEEDS to be *software engineering*. Embedded Software most notably. Also process-control software in factories or vehicles and everywhere where human life or large sums of money are at stake.

On this note I want to pose the **rule (7)**:

*Many people are not aware that such a difference between development and engineering exists (rule 6). However, there seems to be a widespread acceptance that Software "always" contains bugs and later versions will fix it.*

The truth is that this is more of a cultural issue than that it *HAS* to be that way. Better or engineered software just costs more, because it requires different more rare skills. And –unfortunately- the difference between development and engineering is not a black and white one. There is a large gray area, which for many domains may be “good enough” and most cost-efficient.