

A midsummer night's dream of a lazy software engineer

The following is one half prediction and one half hope for a better tomorrow.

Introduction

Typically, a program running on a computer does not know it is being executed or why it is executing. It (the program) is a stream of instructions conceived by its creator to run on a machine. Thus, the program is as “good” as its creator's implementation of the solution to the problem sought to be solved. Since each program is targeted to solve a particular problem, programs are specific and cannot be easily altered to solve a different problem. Modular, object-oriented, functional and procedural programming all strive to create libraries of functions or classes that are generic enough to be used in many differing solutions. If a certain function should act on various types of data, different paradigms have varying approaches to solving this problem – in procedural programming functions with different names are simply implemented or some kind of generic pointer manipulation combined with typecasting is involved. In object-oriented programming one uses polymorphism to do this - the function keeps its name but the “correct” function is automatically called to match the type of data it is to process). However, in both cases the responsibility lies with the software developer to predict all possible cases and make sure things “go as planned”. Finally, each program runs on an operating system (OS) which acts as an intermediary between the program and the processing hardware and memory. The OS provides various mechanisms to allow a program access to machine resources while also allowing other users and programs to do the same, generally and hopefully in egalitarian fashion. Most operating systems create a container called a “process” within which a program executes. The executable program is created using a compiler (yet another program) where the programming language instructions are translated to machine opcodes recognizable by the processing unit and packaged in a form suitable for the operating system to load and manage. The OS will then create a process with certain limitations and privileges, load the program into the process container and turn the control over to the processor chip (CPU). The CPU, in turn, “blindly” executes the instructions, one by one, switching between the operating system code and the user/process code as necessary. The process is generally unaware of this manipulation and so is the CPU.

Note: The responsibilities of solution correctness, completeness, adequacy , security, maintainability and scalability are left to the program designers and developers but also to the operating system designers and developers (and to the hardware designers and developers respectfully). In essence, each level of abstraction is expected to allow certain functionality and restrictions envisioned by its creators.

History of computing has shown us that while software is built every day to solve various problems, it is also riddled with errors and inadequacies. Most importantly, software is treated as an artifact or a product. This (at least) implies that it is not expected to possess awareness of itself or its function (raison d'etre) and consequently, that it cannot behave as an organism, i.e. grow or adapt. The terms “growth” and “adaptation” have different connotations in today's software development cycle and are

generally taken to describe the life of the product as it is being used by its users and developed by its creators.

Now we describe a typical approach to developing software. A problem is defined and its definition is conveyed to the developer. The developer builds a model representing an abstraction of the problem domain, including the understanding of the data involved. Hopefully, the model, being a simplification, did not exclude any relevant knowledge of the problem. Based on this abstraction software is designed and a solution is coded in a computer language of choice. This solution can then be compiled to produce an executable program (compilation step can be skipped in case an interpreted language is used). Regardless of whether the solution is interpreted or compiled, it is ultimately a stream of instructions that is executed on a machine, be it virtual or real.

Note: the instructions executed by the machine are immutable. The functions built on top of these instructions are also immutable. The only way to change anything is by virtue of having the developer rewrite portions of the code and go through the steps outlined above to produce a new version of the software (this usually happens due to debugging needs or simply due to adding more features to the solution).

Definitions

We define a nanite as a device generally of very small physical size, used for medical or engineering purposes where the operational size of the problem-solution pair is on the nano scale. The nanite conducts its work in the real world, such as a human body or crystal structure (from this point on - “the system”) where it possesses the right tools and knowledge to perform a simple task well, if the right conditions are fulfilled. An example would be a synthetic cell injected into the human body to attack a specific virus or bacteria.

It should be obvious from the above discussion that a direct translation of the nanite paradigm to the world of software would be impossible, mostly due to the fact that nanites solve real-world problems while software represents solutions to models of problems as solved by the software developers (the software IS the model and thus represents both the problem and the solution!). The closest concept to the idea of nanites in the computer world are probably various malware artifacts, namely viruses and worms. It is worth noting that while viruses and worms can be viewed as “units equipped with tools and means” and that while such units can be injected into computer systems to perform a function, malware in general is built the same way as conventional software and is still unaware of its presence or purpose. Therefore, it falls under the same limitations as ordinary software artifacts.

Soup, the ultimate goal of this paper exercise, is the incubator for various nanites, a place where they run, grow, couple, die and get recycled. It is a place where the human developer injects a blueprint that then builds on the existing functionality as implemented by the nanites living in the soup or the ones borrowed from another soup (owned by another user somewhere else in the networked physical world?). This blueprint, now called “software”, is a specification of an organism which exists for a

particular purpose – to solve a particular problem. Notice that not all nanites necessarily just execute a function, many of them will represent data. For example, number “5” could be a group of five nanites somehow connected together. Viewed like this, the operation of counting has a contextual meaning and is not just a matter of being seen as storage or type (e.g. float vs long int vs short int). Furthermore, every nanite is bubble-wrapped in its outer layer, the interface. This interface is made up of a pattern of ones and zeros and can be used for coupling to other entities in the soup, if the conditions are right and the interfaces match somewhere on the surfaces of these nanites, for example beyond a certain length of the matching portions. Coupling entities also couples their functions, allowing for behavior modification. For example, one entity can exist for the sole purpose of data output. Another entity can be a sine function implementation. Together, they can couple to print a sine function result to the screen. However, this same sine function nanite can be coupled elsewhere on its surface to a completely different “modifier” nanite whose purpose, for example, might be to calculate a square root of its attached neighbor. There could be a nanite in the soup for any possible function we can imagine and where the soup does not have a desired entity, it can be borrowed from another computer. Nanites are envisioned to be mobile and inhabit various machines at different times or as needed. Consequently, repositories of nanites could be built with the sole purpose of offering true and tested entities for use to others to develop their own solutions to their own problems. The problem of software development is now reduced to providing a blueprint for what coupling (or decoupling) needs to take place and when. The soup will take care of the actual “unfolding” of this blueprint and the “growth” of the solution.

Conclusion

It is not immediately possible to fully apply the concept of a nanite soup to today's computing – the hardware of our time is not built with the notion of a software component being able to affect the physical characteristics of a device (this would open up countless possibilities of hardware being able to “morph” itself to the needs of the software). However, a hybrid approach is possible – an operating system like Linux could be modified to support the notion of the soup and still retain its hardware control characteristics (thus allowing the use of today's “braindead” machinery).

“Growing” software would open up various possibilities not yet available with today's paradigms. For example, we can share Java or C# components but they do not “plug-in” to modify each other as need arises (or perhaps by accident as they co-exist in the same soup?). The environment these components run in is rather constrained and tightly controlled. It is up to the developers or engineers to envision a solution to a problem, implement it and also foresee and control all possible paths of execution. As the problem grows in magnitude or changes in nature, the problem of engineering the solution is not only a matter of providing new or different functionality but also of fitting the new with the old and making sure it all works as expected, a process commonly referred to as backtesting.

Increasingly, programmers and engineers will demand an easier way to control complexity. Beating up on the same old paradigms could eventually bring us to a grinding halt when it comes to developing

software. The body of science today implies that the adaptive organisms survive longer and fare better. Perhaps it is time software is viewed the same way?