Sustainable Software Systems

Creating long lasting software

sus·tain·able /sə-'stā-nə-bəl/
Adjective: Using a resource so that the resource is not depleted.

Introduction

The origin of this paper stems from a synthesis that has taken form in my mind based on my experience with software system life-cycles combined with recent literature on sustainability. As with most progressive work, I was not able to see or formulate an accurate problem statement until I reached out of the daily routine and took on a fresh and complete different perspective.

For a long time, it seemed like the new Agile movement was the answer to everything that was wrong with the waterfall software system paradigm. Alas as years since the Agile inception goes by there's a lingering feeling that there is still a lot of room for improvement in the software lifecycle process. Agility and lean manufacturing concepts certainly seems like the right answer for developing software but leaves a lot unsaid about maintaining it or even keeping it alive.

Why is it then that once a software system has been developed (of course by using Agile/Lean techniques), we simply can't just hand over the "system keys" to the custodians of the new system, erase it from memory and then move on to the next project?

The answer is simple: The 2nd law of thermodynamics.

This law (like many other laws of nature) has a fundamental effect on our universe and everything in it, objects as well as processes. Even human thoughts and emotions are affected by it. In the next section, I'll explain why and how the 2nd law of thermodynamics is so devastating to our software systems. Other sections included herein are, "How software dies", "Hardware is dead, long live hardware", "Misleading metaphors", "The true life-cycle", "Knowledge by synthesis", "Supporting echo-systems", "Nature 2.0 - The Mammal Edge" and "Concluding remarks".

There will be no code examples, nor mention of specific software languages in the text that follows. For a system to be truly sustainable it has to stand the test of time. This sometimes might even involve a complete refactoring into a new language paradigm. The following sections will focus on natural principles and observations of human interactions that is or should be time invariant. As a reference for the remaining of this text, lets define a software system as sustainable if the system's initial feature set still exists after an arbitrary point in time with or without augmentations such that it still provides meaningful results (i.e is still operational):

$$\forall\ n > 0 : F_{t_0}(x) \cap F'_{t_n}(x) \neq \varnothing$$

I.e. a software system is sustainable if for all n, the intersection of the initial feature set F (x) and the feature set F'(x) at time n after the initial time is non-empty)

This definition implicitly states that if the system is still working but it's current feature set have nothing in common with the initial one, it's a different software system altogether and is probably still working due to constant tampering

The 2nd law of thermodynamics

$S' - S \geq 0$ *(s = entropy at an earlier point in time. s' = entropy later on)*

I.e. the entropy in a closed system always strives for a maximum. But what does that have to do with software systems one might ask? The concept of entropy as used in classic thermodynamics doesn't seem to apply directly to software other than the more general observation that without adding energy, any system will become more disordered. The best everyday example of that are my children's rooms. If you move a single object the room would be less messy since you now would have a local cluster of organization, i.e. a state of maximum entropy. However if we make the leap over to statistical mechanics and it's definition of entropy, it quickly becomes more clear. Statistical mechanics defines a macroscopic state in terms of it's microscopic states. Translated further to software systems, macro states are the different states that the overall system can be in (on, off, running, waiting, processing, halted, crashed, etc..) and each of these are defined by the set of micro state values for all sub-processes, memory contents, internal registers, etc...) that can be grouped together to represent each macro state. The entropy of a macro state is defined as the number of sets of micro states that it includes. The 2nd law of thermodynamics applied to software systems then reads: "Unless work is done, a system's entropy will always increase until it gets to a macro state with the highest possible entropy". I.e. if a software systems is not maintained each of the components will in time always reach the most probable macro state (which typically is "not working" since of all possible combinations of micro states most sets would belong to the "not working" macro state). Work in this context represents typical every day tasks that one might perform on a system - backups, delete log files, add memory, fix bugs, add features, etc...
A more direct and popular interpretation would be: Unless someone maintains a software system it will always end up useless. What I tried to show above is that this statement is not only a "hunch", it's one of the most fundamental laws of our universe. There is no "free lunch" when it comes to system maintenance, someone has to do it. The question that remains is who and how? Read on to find out...

How software dies

If the group of architects, developers and testers responsible for implementing a software system are all disbanded, that software is for all practical purposes "dead". It might still be running in a production environment. However it would be more like a decapitated rooster still running around the chopping block. As discussed in the previous section, the 2nd law of thermodynamics guarantees that without anyone around to perform work, the system will sooner or later become useless. A common misconception is that as long as there are lots of documentation covering all aspects of the software system, all one has to do is hiring new people to pick up from the old group when time comes (i.e. when the system starts to act up). Anyone actually being responsible for a system under these conditions knows that this is extremely hard to achieve.

One of the main issues stems from an early observation in 1968 about systems development that is commonly referred to as Conway's Law: *"Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure"*. A more clear way to rephrase this could be: *Anywhere there's a team boundary, you will find a software boundary*. From this observation it is easy to understand the similar phenomena "Not Invented Here", i.e. the tendency to rather do the work from scratch within the team instead of (re)using a solution made by an outside group. These two concepts that predicts intergroup behavior are based on teams existing in the same time frame. However in the scenario of an abandon system there is also a time-shift involved, i.e. the disbanded group is no longer available to interact with (other then through the document artifacts left behind). The tendency to rather do it from scratch internally instead of trusting existing solutions made by the previous team is still equally true. Particularly if the documentation left behind is hard to comprehend. Which typically will be the case if it was produced by the previous team members who might have made a few too many assumptions about the level of preexisting knowledge on behalf of the reader.

There are of course a number of other circumstances that would more or less kill a software system even with a sufficient number of existing team members around. Dependent software or hardware resources becoming obsolete. Thanks to Moore's law this is an ongoing process and must be planned for by allocating resources (financial and human) in time to perform any necessary upgrades. Lean manufacturing advocates the concept of not making a final choice until the latest responsible point in time. However, that concept is more applicable as a guide for making the final cut when moving forward with multiple solutions (i.e. set based design). In this scenario it is more useful to define a point in time when the upgrade must commence in relationship to other known or estimated points in time and durations:

t = point in time when upgrade must start.

t_{wrk} = duration of upgrade work.

t_{ord} = point in time when new hardware / software must be ordered.

t_{del} = duration for new hardware / software to be delivered.

t_{obs} = point in time when old hardware / software becomes obsolete.

t_{err} = duration of margin for error during the upgrade process.

t_{dec} = point in time when the decision to upgrade must be made.

t_{prep} = duration of preparation work between t_{dec} and t.

Since we are mixing points in time with durations of time we can express points in time with the corresponding epoch value (i.e. seconds since 1/1/70) instead of a date and durations as seconds. Then we get the following basic expressions:

$$t_{dec} < t_{ord} < t < t_{obs} \quad ; \quad t_{dec} + t_{prep} < t \quad ; \quad t + t_{wrk} + t_{err} < t_{obs} \quad ; \quad t_{ord} + t_{del} < t$$

So the appropriate time t for any upgrade work to start is one that is true for all of the above expressions. In the section "Knowledge by synthesis (not by fact enumeration)" I will explain in more detail why documentation alone is insufficient as a vehicle for knowledge transfer.

Hardware is dead, long live hardware

One of the biggest breakthroughs in computing during the last couple of years is the emergence of the Cloud and more specifically the manifestation of it that could be called Platform-as-a-Service (PaaS). Software-as-a-Service (SaaS) has been around for a while, but is merely a preview of the true power of the Cloud. The main reason why the SaaS concept never struck big with large companies is that most such offerings are provided as a black-box and shielded from the rest of the organizations IT-infrastructure (i.e. not very useful). However, just like many other buzz-word technologies (AI, O-O, Patterns, etc...) it's now part of our every day computing environment (gmail, Google Docs, Google+, Facebook, twitter, etc). The largest struggle for large companies (and startups too for that matter) is the sprawling cost of IT-administration. As the need for processing power and data storage increases with growing success, so does the need for administrators that can tame the growing fauna of specialized servers, network infrastructure and worst of all, an intertwined mix of custom and off-the-shelf software that is in constant need of version upgrading and integration.

In the view of the effects of the 2nd law of thermodynamics as described earlier, together with an understanding that with more computational nodes involved in any given process, the more fragile the system will be. We will sooner or later reach the conclusion (and experience it empirically), that the more mixed technical infrastructure we acquire, the deeper down the rabbit hole we get. To understand why an increase in system nodes makes for a more fragile system, let's turn to basic math:

Assume that we have 6 systems involved in a given process (e.g. client, network infrastructure, application firewall, API server, business object server, database server) and each system has a probability of "working just fine" equal to 90% (or 0.90). Then the probability for the total process chain to "work just fine" is equal to: 0.90 x 0.90 x 0.90 x 0.90 x 0.90 x 0.90 = 0.53 = **53%**

Enter Platform-as-a-Service. What this really does for companies is that it completely abstracts away the notion of hardware. Gone are forever the notion of hardware nodes, server names, IP-addresses, environmental/configuration variables, server-server communication, etc.. Instead there is only "the software eco-system" which best can be described as an limitless scalable orchestration of distributed software functions deployed on the internet (i.e. SaaS or more in vogue RESTful APIs). The orchestration of these SaaS/RESTful APIs is done in a high level langue supported by the PaaS provider. The greatness does not lie in the fact that a modern high level language is being used, the greatness lies in the fact that the software no longer is targeting a specific server or cluster of servers, in fact the software is oblivious to the concept of servers. Instead the software deals with interfaces whose most specific addressability is not a hardware node but an internet domain. No matter what the service need is (e.g. web-service, geo-location data, file contents, database records, etc) it's accessed through an internet available URL. Not even when deploying a PaaS implementation will server names be exposed to the developer, mainly because it's completely non-deterministic.

So why is this so special? Because looking at the statistical mechanics entropy model, there are far less micro states involved for each macro state with the consequence that the set of all micro states for any given “failed” macro state has orders of magnitude fewer micro states than for a system made up of a plethora of mixed server nodes. Thus making it much, much more stable when ignored. Yes there’s still lot’s of hardware involved, but it’s completely removed from the sphere of influence and control of the implementing organization and are instead taken care of by someone else which have been smart enough to create a huge scalable distributed network of cheap homogenous commodity servers that can crash and be replaced by a very small group of administrators during run-time without anyone noticing.

It’s important to understand that the concept of server virtualization is not the same as PaaS. It will actually makes things worse by introducing even more micro states for each macro state (e.g. more server names, ip-addresses, environment/configuration variables, etc...). The winning concept is still the magic trick of abstracting away the concept of hardware altogether. It’s almost like we traveled back in time to 1960 and experienced mainframe computing all over again, except these born-again mainframe software systems do not run on hardware and have endless scalability.

Misleading metaphors

Metaphors are very helpful whenever we want to quickly explain or understand a new concept. The reason for this is that the human mind really is only a pattern matching, hierarchical classification machine that loves puzzles. Experiencing an old domain with a new twist or a new domain with references to a known old domain allows the brain to eagerly synthesize new patterns that can be added to its previous structures. Being presented with a complete green field domain is tougher, since there might be no previous structure to build upon. However if the chosen metaphor is incorrect or misleading, irreparable damage to how our mind perceive things will occur. A great example are the two most commonly (mis)used metaphors for software systems - the “Factory” and the “Engineered Structure”.

What’s wrong with the “Factory” metaphor?
This metaphor is not entirely wrong, but believing that implementing software is like running a factory is plain wrong. Implementing software is like building a factory including designing and building all of the machinery that the factory needs to manufacture products. But as any designer of factories will tell you, building a factory is an “Ad hoc” process where no solution looks like the other (i.e just like implementing software). Actually using the software once it has been put in production is like running a factory, so this metaphor is slightly off and offers no help at all in better understanding the software development process.

What’s wrong with the “Engineered Structure” metaphor?
The “Engineered Structure” metaphor has similar issues as the “Factory” metaphor. It suggests that implementing software is an entirely deterministic and repeatable process where all design decisions can easily be looked up in engineering charts and tables. Alas the biggest fallacy of this metaphor is that it suggests that software is a static construct, much like the mechanical machines, buildings or other common civil engineering structures that this metaphor is based on. Not so - most software is created to support a business organization and is under constant pressure to change and adapt to an ever-changing business climate. Modern day business processes are better described as complex adaptive systems (CAS) rather than static flows of information and decision points.

So is Complex Adaptive Systems the correct metaphor then?
Not really, much like how the factory metaphor is better suited to describe software systems running in a production environment, the CAS metaphor works best when evaluating the progress of a software system in relationship with the business process it supports. Read the next section for a more natural and better fitting metaphor as far as understanding the software implementation process.

The true life-cycle

We are now ready to formulate a better way to look at the software process from a sustainability perspective. Just as we have used laws of nature earlier, its time to once again look to nature to discover a process whose longevity will make even the most long lived mainframe implementation look like the new kid on the block; the growth and decay cycle of organic matter powered by photosynthesis. In it's natural form it has been around since the formation of organic matter. In it's human controlled form it has sustained for as long as 10,000 years (some rice paddocks in China have been around this long). We also know what will happen when the basic structure of this process is being violated - extinct cultures, soil erosion and formation of sand deserts. There are profound learnings to be made from this process once we formulate an appropriate level of abstraction so it can be applied more generically outside the domain of agriculture.

First a very basic observation: In our western society we tend to think about growth and decay as the starting point and the absolute end of a process, mostly because we have a hard time dealing with the concept of death. But once one studies this process more closely it becomes clear that this is a short term, repetitive cycle, which has more do to with replenishing and rejuvenating that the finality of death. The decaying organic matter left behind after harvest acts as a catalyst for replenishing the soil with nutrients thanks to the process of making the layer known as humus. Another basic observation is that nature and the world's cultures are defined by cycles of seemingly opposing forces: self-organization/2nd law of thermodynamic, life/death, day/night, growth/decay, build/destroy, work/rest, yin/yang, etc... It seems like there must be a very basic and profound pattern at work here.

So how do we move from our traditional point of view of software implementation having a project kick-off followed by lots of hard work ending with the deployment to production?

It's easy - if we look at any of the software projects that we have been involved with a longer period of time, we'll discover that it is a cyclic process. It's just that compared to the first implementation cycle, the effort involved in consecutive cycles is orders of magnitude less. Perhaps one approach to better software sustainability would be to do less but more often, while also making sure that resources are replenished between cycles?

What does it mean to replenish resources and what happens if we don't?
Another important aspect of the growth and decay cycle in nature is the occurrence or lack of diseases in the soil, crop, livestock and humans. What patterns do we see in nature and how does it translate to software? When and why do nature thrive v.s. suffer as far as diseases are concerned. Again, this is no small matter, there's a reason why we have been plagued by E.coli, mad cow, etc recently and there are certainly lessons to be learned from it.

So by using nature as a metaphor we will try to better understand the following concepts: How to better organize and plan work, how to keep resources replenished and how to deal with defects.

To be more in sync with nature, look to existing practices of Agile iterative development but also the new trend of continuos deployment. Shorter cycles of intense work with periods of replenishment between. This doesn't necessarily mean rest, more like "sharpen the saw" as defined by Stephen R. Covey in the book "7 habits of highly effective people". Making sure to perform tasks outside of the daily routine to keep energetic and enthusiastic about starting on the next cycle as well as investing in appropriate software/hardware tools and reference materials.

Another key observation from nature - mono crops will always end with erosion, no matter how much chemical fertilization and pesticides that are added. I.e. make sure that team members don't work on the same tasks over and over again. Adding "chemical fertilization" and "pesticides" like tracking tools, time management systems, computerized Agile management, hovering managers, threats of termination, etc.. will not help in the long run. Better to trust that people will actually do what's best for themselves and the team by respecting their skills and making sure they have a variety of tasks to work on. Daniel Pink's book "Drive" is a great reference in understanding what truly motivates people. The first deployment of a software system should be one of many future steps, preferable with an initial feature set that is still meaningful to the business. The same team can then (with some variation over time in staffing) continue to work on the system or better yet, multiple systems to avoid macroscopic "mono crop" fatigue.

Natures solution of dealing with disease is spelled abundance. There are no natural processes that specifically targets disease in nature, rather by providing an abundance both in amount and variety of nutrients, raw materials and fauna, major outbreaks of any disease is naturally avoided. When humans disrupts this state of abundance, diseases will inevitable follow. The key learning for the software process - don't try to get away with minimal resources, either in the level of knowledge or amount of people. Of course no company can afford a hoard of software Ph.d's on their projects, but making sure to have a mix of people with lots of experience and/or education will certainly payoff compared to hiring too few developers having only rudimentary knowledge of just one programming language and no other skills. No matter how much "pesticides" management tries to add to stave off "bugs" and delayed milestones.

Nature does provide a defense mechanism for outside threats, sort of…
Any sufficiently developed organism (e.g. humans), can interpret natures own early warning system; e.g. aposematism, bio indicators (e.g. mosses or Lichens) or indicator species (e.g. Canaries).

Similarly, software systems should also utilize this strategy; e.g. automated tests, automated production diagnostics ("phone home") or click-stream analytic processing.

Knowledge by synthesis

In this section we'll look at how humans actually learn (a key to sustainability). A truly sustainable software systems should be able to last for generations. This statement is less ridiculous than it seem in the light of some of the mainframe systems that are still around today. As long as the main feature set remains similar, one can claim that a software system has sustained even across major changes such as switching operating system, implementation language or database management system. Based on the section "How software dies" earlier, we know that the best way to sustain a software system is if we at all times have team members around with previous experience of it. This doesn't necessarily mean that we need to have all of the original team members around (or any for that matter), rather there has to exist a tradition of keeping the collective theory of the system alive as well as passing it on to new team members as they join in. A software system can not easily be understood simply by reading documentation. The only/most effective way of gaining understanding of a software system is to combine system interaction with shared knowledge from other team members periodically dispersed at a pace that the recipient can digest without being overloaded. In order for anyone to be able to correctly modify a system without unwanted side effects they need to be able to (at some capacity) simulate its behavior in their own mind. This can't be done without true understanding of the system, which means that the mind needs to (re)create a mental model of the system's feature set as well as being able to simulate this at some level of abstraction that generates results similar to running the system. No two team members will ever have the same mental model of the system but as long as they can simulate its features with the same simulated outcome, one would have too agree that it's sufficient. Typically the process of creating any mental model works like this: Formulate an initial naive model and validate it with example features and data sets based on system discovery (i.e. using it, reading about it and through peer collaboration). If the simulation doesn't generate correct results, modify the mental model slightly so it works for the failing example set. Iterate this process until satisfactory results are generated (or give up). Based on this description one can easily see that knowledge is created (synthesized) by lots of iterative examples rather than trying to understand at once a predefined explicit model defined by lots of detailed drawings and/or text paragraphs.

One helpful tool would be if the software could be "probed" in run-time and be made to generate an audit/"breadcrumb" trail that could be mapped against the source code. It's very hard to predict software behavior by inspecting its source code only. It was even proved as early as 1936 by Alan Turing that no software (Turing Machine) could exists that could predict the outcome of all other software (some, yes - but not all). This is popularly referred to as "The Halting Problem". Such probing would also be very helpful when resurrecting "dead" software. The simplest implementation would produce logs in a log file with alphanumerical markers easily found by inspecting the source code. A more advanced solution would include a data collecting mechanism that would act as a "flight recorder" during execution and then stay in memory so it could be interrogated about the software structure and probed for check-points easily mapped to source code.

Supporting eco-systems

Just as in nature, a process cannot be successful without support. In the realm of software systems there are many interacting tools, processes and organizations, all important to a system's sustainability. This can be described as an intricate network of interests all feeding of each other in some way. Here I will point out two specific support systems that are easy to ignore but can be crucial to a systems survival: automatic test harnesses and crowd participation.

Automatic test harnesses
The role of an automatic test harness is not to tell us whether a developer's most recent contribution is working or not. The real reason is for it to act as an insurance policy down the road, perhaps 6 month or 6 years from now. It will automatically tell us if a recent change to the system mistakenly violated any existing functionality that so far has been working fine. It is also a great tool when making a major upgrade to a system as far as changing operating system, implementation language or data base system (hint: upgrade the test harness first). It also makes a lot of sense to augment the test harness with the type of software probing functionality that was defined in the previous section.

Crowd participation
This is the notion of treating the users as a part of the system. Most systems can not grow past its initial data set and reach critical useful mass without the exponential growth enabled by letting the crowd into it's content creation and curation. It will always be a fine balance between too much and too little control of what the crowd can do. Either one can be devastating to a system.

Nature 2.0 - The Mammal Edge

The comment earlier about nature not having any defense mechanism against diseases is not entirely correct, mammals have more or less advanced and independent immune systems to help stave off outside threats. Translated to software this can be an invaluable approach for any hard to deploy scenarios, e.g. software running on unmanaged mobile devices. There are numerous ways an unmanaged mobile device can suffer a software issue/incident that couldn't be detected as part of testing (unless you have a NASA budget). Building an independent error detection and correction layer could make a lot of sense. Make sure that such a layer is truly independent of the underlying failing feature so that it can act as a backup system in case the primary one indeed turned out to be faulty.

Another one of nature's clever strategies becomes evident when looking at the development of the mammal; starting with the Zygote all the way to the newborn we see that nature is not aiming for the final result right away. Rather it iterates through many stages to reach the final product. Applied to software development this means that it is ok to start with a very simple implementation (sometimes called Minimal Viable Feature) and then iterate/refactor until a more complex final solution emerges. This approach works equally well for intermediate implementations to reach a final feature that seems too complex to approach in one go, as for features that will be released to users in it's simpler form and only later mature into a more advanced manifestation.

Mammals also have one of the most extraordinary devices ever created either by man or nature, the neocortex. The neocortex as manifested in humans holds the key to building truly intelligent systems. Looking at Computer Science in general and Artificial Intelligence in particular we have so far been going down the wrong paths altogether. Neither one of the two major branches so far, decision trees and neural networks, have managed to capture the intricacies of the human neocortex. Creating decision trees in all it's form (manually, programmatically, from examples, etc…) is not even close since it's still very much an algorithmic approach (which the neocortex isn't) and most attempts at creating neural network so far has either been too simple or ignored the most important feature: having an hierarchical structure. The sooner we can get away from the current trend of massive (but dumb) processing of "Big Data" and start focusing on cracking the code on how to best simulate the human neocortex, the better. Any attempt at such a simulation should at least contain the following elements:

- Hierarchical structure.
- Invariant representation.
- Auto association.
- Temporal sequencing.
- Hebbian learning.
- Forward and backward propagation.
- Inhibition.

Concluding remarks

Time to wrap up the sustainability bag. The topics discussed so far might seem like random banter but there is an underlying theme throughout. There are ubiquitous laws of nature in play that effects all and everything. From formation of galaxies to the signal paths in the human brain. At its core nature is a zero-sum game between opposing forces simultaneously acting on both a macroscopic and microscopic level that affects everything around us. From planetary movements to software system design. Ignoring this and only take into account the latest man-made paradigm that happens to be in vogue will always be a mistake. In the end the 2nd law of thermodynamic will emerge as the final conqueror alas without any remaining audience to celebrate the victory.

Until then whenever in doubt, push away from the keyboard and ask yourself:

“What would nature do?”.