ExxonMobil

Do you know how to KYSSS?

(Keep Your Sci. Software Sustainable)

a tough love story

Vadim Dyadechko

Disclaimer (just in case)

Certain parts of the presentation may look to you:

- too obvious
- too general
- too specific

Well, the audience is too diverse, hopefully everyone will find something useful.

Some statements may sound over-simplified or arguable;
it was done on purpose,
to make the message unambiguous
and spark a discussion.

I understand that the right answer to most real-life questions is “It depends”;
unfortunately, this answer does not help to stay focused
and deliver quality products on time & on budget.

The opinions expressed here are solely my own
and do not express the views or opinions of my employer.

Right to the point: defining “sustainable s/w”

Wikipedia:

“Modern use of the term sustainability is broad and difficult to define precisely.”

Google:

“*able to be maintained at a certain rate or level”*

Merriam-Webster:

“*capable of being maintained at length without interruption or weakening”*

Cambridge:

”causing little or no damage to the environment and therefore able to continue for a long time”

Associations of “sustainable”:

- long-lasting
- productive, robust, resilient
- manageable
- low-maintenance, affordable
- environment-friendly

Does such a package even exist?

Right to the point: defining “sustainable s/w”

How would Alan Turing define s/w sustainability?

A scientific s/w project is sustainable if

it consistently meets expectations of all stakeholders:

users, developers, sysadmins, management, etc.

As you well know, s/w does not always generate positive emotions:

- s/w projects have notorious record of delays, over-budgeting, and cancellations;
  the larger the project the higher the odds of failure
- large s/w systems are known to be slow and glitchy (&^#&%#&%&!!!!)
- maintaining a large s/w product is stressful…
  …unless the product eventually becomes “too large to fail”

One may argue that all large projects have similar problems.

True.

But the s/w development stands out in a crowd.

Bits of Brooks's wisdom (hint: look for the subject)

"The Mythical Man-Month":

"Adding manpower to a late s/w project makes it later"

Chronic infection:

A suitably complex system has irreducible number of bugs (bug out → bug in)

Snowball effect:

"How does a large s/w project gets to be one year late? One day at a time!"

"The second-system effect":

Elegant prototypes are succeeded by over-bloated production systems (due to inflated expectations, complexity, and confidence).

"No silver bullet":

"There is no … [s/w development] technology or … technique, which … promises … order of magnitude improvement … in productivity, in reliability, in simplicity."

Illusive nature of s/w reflects human mentality

While s/w is stored and run on computers,
it is developed only by human brains.
Software is illusive because it is a mental product.

"It's all in your head" means *"You are out of touch with reality"*

- Intangible objects look to us less constrained than the real ones
  (think of a teenager with a new credit card)
- Our ballpark estimates always ignore the "distribution tail"
  (we tend to estimate the total of a long shopping list as the sum of large items)

How can we expect to create a perfect s/w product
if we constantly fall into the traps set up by our brains?

Human factor: “Here be dragons”

Human brains are not wired to handle complex s/w systems;

we are not always aware of our physiological limitations,

and too proud to admit that we are not in full control.

We constantly expand our capabilities

by improving the code structure

and automating development process,

but our appetites grow way faster than our powers. (“No Silver Bullet”)

We tend to have *unrealistic expectations* about s/w,

are *over-optimistic* in our plans, *systematically* under-estimating

- the development / testing / integration / operational cost
- the complexity of code and stability of operational environment

Awareness of our own fallacies

is the first step towards sustainable s/w development.

Managing expectations: hardware vs software

“Why is it so much harder to manage software projects than hardware projects?”
(Thomas Watson Jr., IBM)

We don’t talk about hardware sustainability because:

- hardware is tangible
- hardware is final (static)
- hardware is fit for purpose
- hardware is well documented
- hardware has well defined lifetime
- hardware has well defined operational environment:
  interfaces, protocols, tolerances, etc.
- hardware quality is taken very seriously (prohibitive cost of failure)

Hardware is quite rigid, we accept it and deal with it, no complains.

Because we know that the laws of physics assume no tolerances, the reality is simply unforgiving.

Managing expectations: hardware vs software

Software, on the other hand, is perceived / expected to

- be intangible
- be flexible and customizable
- accommodate new features
- be fairly portable, work on a wide range of platforms
- be compatible with other pieces of software
- be of low maintenance
- eventually become bug-free (well, usable at least)

Unfortunately, a lot of these expectations are over-inflated.

Managing expectations: hardware vs software

Here's a quick reality check:

- most production s/w designs are surprisingly rigid
- compatibility requires fair amount of fitting and testing
  against components you do not control
- portability complicates the source code, build framework,
  multiplies the cost of testing (how badly do you need portability?)
- every little feature increases overall complexity (snowball effect)
- complex codes can only be maintained by their authors
- complexity breeds and shelters bugs (chronic infection)
- complexity hogs time

Every wish has hidden cost, frequently substantial, sometime prohibitive.

But it does not stop us from requesting / promising "full service".

Such is the irresistible magic of word "soft".

Managing expectations: be realistic = pessimistic?

To a large extent, the s/w sustainability problem is a problem of our unrealistic expectations.

A: "Oh, come on! You can't fit into this costume!"
B: "Is it Lycra?"
A: "Yes, but it's still governed by the laws of physics." (Just Shoot Me)

Since we don't know how far we can "stretch Lycra" without tearing it,
and we are known to have optimistic bias,
it is safer to wear a pessimistic hat (well, helmet) all the time.

We are much better at managing real objects, let's manage software as such:

- settle for a fit-for-purpose specification
- be conservative in expanding the scope
- take quality seriously

(non-essential) complexity is your enemy #1

- **Each system is only as robust as it is simple**
  - learn to be principled about requirements
    compromising code simplicity for incremental benefits
  - minimize the number of custom config files and options,
    introduce smart defaults instead

- **The key to success is self-imposed sharp focus:**
  - stick to one OS (Linux)
  - stick to one compiler suite: GCC, LLVM, Intel, PGI -- your call
  - stick to mainstream hardware (multi-core CPUs)
  - stay away from multi-threading (MPI only)
  - stay away from high-maintenance 3rd-party dependencies
  - stay conservative in terms of tools/standards (make, c++11, perl5, python2.7)
  - resist the temptation of writing a general-purpose library (3x cost)
  - explicitly limit your support liabilities

Post-release support and refactoring: part of the package

Both code and operational environment are constantly evolving

- One cannot write a piece of code and leave it alone;
  it will eventually become obsolete/unusable
- S/w requires ever increasing support effort, which is not popular:
  - perceived to be secondary to coding
  - viewed as a damage control overhead
  - creating new bugs is more fun than eliminating the old ones
- S/w also requires timely re-factoring:
  - relieves the “internal stress”
  - allows to control complexity of the project

Post-release support and refactoring are unavoidable and consume fair amount of resources; plan and execute them accordingly.

Testing and deployment: part of the package

IT operations (system, build, integration, automation):

- viewed to be secondary to coding
- “negative” visibility (sysadmin curse):
  - smooth operations generate no positive feedback
  - occasional screw-ups generate negative feedback
- hard-to-measure deliverables
- highly scattered unstructured subject knowledge

“They don't teach this stuff at school.” (Alan Wild)

DevOps:

- **testing and deployment are integral parts of development cycle**
- short release cycle
- streamlined operations
- highly automated QA/QC and deployment processes

How to Keep Your Sci. Software Sustainable

- **Start with yourself: beware of human brain fallacies**
- **Set the right expectations: manage software as a tangible product**
- **Give tough love: fight complexity, keep focus sharp, stay conservative**
- **Prioritize support, automate testing and deployment**