Random Thoughts on Software Sustainability

Vadim Dyadechko

Disclaimer

Some statements below may look over-simplified or provocative; it was done on purpose to make the message unambiguous and spark discussion. I am well aware that the right answer to most real life questions is "it depends". Unfortunately, this answer does not help us to stay focused and deliver products on time and on budget.

Human factor

Human brains are not wired to handle complex (software) systems. Unfortunately, we are not always aware of our own limitations. We constantly expand our capabilities by improving the code structure and automating development processes, but our appetites grow faster than our powers. We tend to

- misunderstand the nature of software
- have unrealistic expectations about software products
- be over-optimistic about everything, systematically under-estimating
  - the complexity of code
  - the development/testing/integration/operational cost

Accepting our fallacies and controlling them is the first step towards sustainable software development.

Managing expectations

We don't talk about hardware sustainability because

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

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

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

- be intangible
- be highly customizable/flexible
- accommodate new features, embrace new types of hardware
- last over a long period of time

- be deployed on a wide range of platforms
- be inter-operable with other pieces of software
- include bugs (bugs are tolerated, eventually fixed)

Unfortunately a lot of these expectations are over-inflated; here's the reality check:

- most software designs are surprisingly rigid
- any bit of customization increases overall complexity (snowball effect)
- there is a natural cap on how large a reliable system can be
- portability complicates source code, build framework, multiplies cost of testing
- inter-operability requires a lot of fitting/testing
- code readability/maintainability has low priority with constant time pressure

To a large extent, the software sustainability problem is a problem of our unrealistic expectations. By being honest with ourselves and embracing the fact that software should be treated as a rigid tangible object, we can noticeably improve our chances of success. As well as the budget of development/operations. There is no free lunch.

Keep It Simple

Each product/system is only as robust as it is simple, (non-essential) flexibility is your enemy #1:

- learn to be principled about requirements that compromise code simplicity for incremental benefits
- eliminate (minimize) custom config files/options, introduce simple and transparent set of rules instead

The key to success is self-imposed sharp focus:

- no multi-platform support (Linux only, no Mac, no Windows)
- stick to one compiler suite: GCC, Clang, ICC – your call
- stick to mainstream hardware (multi-core CPUs)
- stay away from multi-threading (MPI only)
- stay away from high-maintenance 3rd-party dependencies (Boost)
- resist the temptation of writing a general-purpose library, it is significantly harder than writing an app
- stay conservative in terms of tools/standards (make, c++11, perl5, python2.7)
- explicitly limit your support liabilities (i.e. the lifetime of each maintenance branch)

Moving target

A fair share of software development effort falls into a category of contingency events only because both code and operational environment are treated as static rather than evolving objects. One cannot write a piece of code and leave it alone; it will eventually become obsolete/unusable. Software requires constant support effort, which is not popular:

- perceived to be secondary to the development
- viewed as a hard-to-measure damage control overhead
- creating new bugs is more fun than eliminating the old ones

Refactoring of production code is disruptive, requires strong political will, and in many places is considered to be the last-resort measure. The post-release support and refactoring toll project resources and feed the atmosphere of delays and over-budgeting unless they are embraced as integral parts of development process and planned for/executed accordingly.

Modularity

The standard "divide-and-conquer" approach towards managing complexity faces serious challenges in HPC world: it is impossible to decouple algorithms from target hardware. The performance race incentivizes monolithic vertical designs, the traditional software stacks with portable body and lean hardware-specific base are more complex and/or less competitive.

The legacy codes have complex history, may not be well structured, may rely on suboptimal numerics. Beyond that, it is not uncommon for the subject area knowledge, discretization, and solvers to be unnecessarily entangled ("physics-based preconditioner", "finite-element solver", "sequential formulation"). Imposing clean overall structure, separating the concepts, and revisiting math are frequently more important than propping code for new type of hardware.

Other thoughts

- Resist the hype of cutting-edge technologies, "sustainable" is the opposite of "fashionable" and "experimental".
- The fit between existing apps/algorithms and new hardware should be an important factor in a hardware purchase decision. The advertised specs look less appealing in the prospect of substantial porting overhead and low hardware utilization.
- Involvement of developers in daily production operations has enormous positive impact on the software quality/usability. The best pieces of software were created for personal use (Unix, C, Perl, TeX, Git).
- Static user documentation is mostly dead: searchable wikis/forums proved to be low-maintenance fit-for-purpose knowledge bases.