USC

USC Viterbi
School of Engineering

Practical Aspects of Software Life-Cycle: Decay, Ivory Towers, and Other Topics

Reza B'Far

Topics

- What are the topics?
  - Software Decay
  - Perils of Agile
  - Architectural Dependencies and Reusability
  - Architecture, Ivory Towers, and Execution
- Introduction: Why are these topics important?
  - Seldom discussed – Organizational "Taboo's"
  - Strategic topics – businesses are more and more focused on short term results, often ignoring or postponing medium term and long term consequences.
  - Immaturity of Discipline – Software engineering & computer science are still quite young comparing to other engineering (mechanical, electrical, etc.) disciplines. The theory in these areas have yet to be fully developed.
  - Risk impact – complexity of the topics create risk that is often "hidden"
- Disclaimer:
  - These topics are not well-researched yet so some of the conclusions are subjective. Great topics for your work on your Ph.D. ☺

What is Software Decay

- What is it?
  - Also referred to as "Software Rot", "Software Erosion", or "Software Entropy", Software Decay is essentially an empirical observation that all software systems eventually first become legacy systems and then become extinct.
  - Given the youth of the software discipline in general, the reason(s) for software decay are not perfectly known, but we do have some insights to why it happens.
  - Much like radioactive materials were prior to discovery of strong and weak nuclear forces, we detect software decay and rate of software decay mostly through observing the actual decay. By this time, the decay has already happened.
  - Software Decay tends to be:
    - An exponential decay, decaying rapidly at first, then slowing down and asymptotically converging, but never reaching, no decay. Eventually software decay can cause software rewrite or "oblivion" (where software is either end-of-life'd and discontinued or it lives on some limited number of systems, with the usage of those systems themselves decaying, but never reaching 0 users.

What is Software Decay

- What determines the rate of decay [subset of culprits]?
    - System complexity seems to be at least a linear driver (if not an exponential driver) of decay. The more complex the system, the faster it decays. Why?
        - Complexity prevents on-going refactoring and clean-up
        - Complexity and knowledge diffusion (how many people in the organization know about something) are at inversely proportional (at best and at worst some decaying exponential relationship). The more complex software gets, the more difficult it is to understand it for the individual members and to scale the organization.
            - Example – Declarative programming – some things get simpler, but debugging and a host of other things become more complex so overall system complexity increases with declarative programming, hence having the reverse effect that it was intended for
    - Number of dependencies between architectural components (both static and dynamic dependencies) seems increases the rate of decay.
        - Incidentally, because of this, code-reuse ends up typically having a negative effect on software decay. So, it becomes a balancing act.

What do we do about software decay?

- What are the ways we have tried to address software decay?
  - Build "Disposable" software: Agile and similar methodologies have largely thrived on selling the value proposition to business decision makers that "for applications that do not carry high liability, etc. in cases of failure, then treat software as something disposable". What does this mean?
    - Build it fast, largely disregard architecture, focus on just the use-cases you need
    - Refactor frequently regardless of size of refator (so, this is the disposable part), but that's OK because it was cheap to begin with! And rebuild based on changing requirements
    - Disposable software works well for applications that will not live long anyways (prototypes, rapidly growing and then disappearing markets, etc.)
    - Disposable software sometimes lives beyond its intended life, and then you have real problems. This is a peril of using Agile.
  - Reduce the size of code with tooling and frameworks. This works fairly well for some period of time until there are disruptions in the market (e.g., Internet replaces proprietary WAN's so COM/DCOM distributed applications become obsolete altogether)

What do we do about software decay (continued)?

- What are the ways we have tried to address software decay?
    - Put in place processes that use tests, round-trip engineering, and other process-oriented techniques to keep the architecture, code, and the actual usage of software constantly up to date (type of work that you have learned in 577). This is fairly effective, but the danger is that building software can (doesn't have to) become a slave to the process in which case you'll end up losing the most talented engineers. Solving technical problems is what keeps great engineers engaged and productive and if they get drown in process, you lose them
    - Focus heavily on architecture: good thing to do, but tends to slow things down and create ivory towers.
- The sad truth: We don't really know how to deal with Software Decay. We are learning there are many factors that drive it, but, much like little-known diseases, we first have to understand all of the drivers for it before we can come up with solutions that address it. And then, we have to get buy-in from the industry (people who work as programmers, software engineers, their managers, and executives) that it is actually useful to spend time and money avoiding decay.

Deep-Dive on Perils of Agile

- Agile let’s you build things fast, but…
  - Pushes the notion of “disposable software”
  - Organically de-emphasizes architecture and design
  - The heavy focus on “just fulfilling the user stories” tends to organically put non-technical people in charge of making technical and architectural decisions whose ramifications they do not fully understand
  - “Build it fast, you’ll have plenty of time to refactor… but some time later… wait, I need you to do something else, why doesn’t that damn system scale? Why can’t I just keep adding features to it at linear cost”?
  - It’s much like “let’s build a house… but we’ll just think about the first room, then once we’re done with that, we’ll think about the second room, etc.” Sometimes it works out (if the house is small), but if you’re building a big house, good luck with that!
  - Very few success stories (comparing to other methodologies) when looking at building large and complex systems.

Architectural Reusability and Dependencies

- What is a dependency?
  - Lots of different types of dependencies –
    - Compile time dependencies
    - Configuration dependencies
    - Run-time dependencies
    - Others
- When you reuse code, you are automatically creating a dependency
  - This DOES NOT MEAN REUSE IS BAD. REUSE, IF DONE RIGHT, IS ALWAYS GOOD.
  - However, it means that code reuse or system reuse is NOT FREE of cost or without trade-offs.
  - Important tenants –
    - Reuse at a high-grain (for example, if you're building your application on reusing cloud-based services, don't pick 10 different services, each of which only offers you one feature).
    - Reuse "behaviorally" versus "structurally"

Architectural Reusability and Dependencies

- Important tenants of reuse (continued)
  - Behavioral reuse tends to live longer because its based on (perhaps through degrees of indirection) HOW users use the system versus WHAT it is that the users use. Incidentally, this is giving a rebirth to functional programming in cloud-based integrations.
  - Understand the technical cost of reuse: every abstraction that you create means you need more memory, you have more function calls, etc. Performance is inversely proportional to reuse
  - Be sure to have not just unit tests for each component, but also integration tests for every 2 components where one is dependent on another or reuses the other.
  - Use techniques such as “Design by Contract” (see Bertrand Meyer) to draw clear lines between the provider and the consumer of reusable code or service
  - Be sure you don’t force a round peg into a square hole just for the sake of reuse
  - Be sure your management understands that reuse doesn’t mean a particular feature is magically integrated in. Make the down-sides of reuse (as well as the benefits) well known.