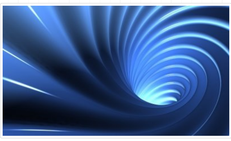
The V2 Vortex



A lot of companies enter, at a certain moment in their life, into the "V2 Vortex". In this article, we're going to detail the pattern of the harsh replacement of the V1 of the application by the V2.

The Case of Software Companies

For a lot of software companies that enter into the V2 Vortex, the history is similar: the V1 of the product was created long ago, for instance in the late 90s or in the early 2000s. It was a success. The company found its market. It has customers. All is great except that in 15 years, technology changed. And so did the competition.

In those software companies, we can often see a lack of software culture, and often, business-oriented people surfing on a tool that developed and gathered success. The R&D team is generally composed of IT people that preferred implementing new features for customers rather than making the development process change progressively to modernize slowly but surely the product.

The problem looks like a pattern: after a large amount of time, even the best buildings get old when they are not maintained. With years, it becomes difficult to implement new features; evolutions are slower to much slower, and the software bugs are more and more complicated to fix [1].

Software is a complex activity because when things go wrong, the more customers, the more troubles, the more time and money spent in support, the less money to invest and the less time to fix. This is the dark side of the software moon, the one that enables selling plenty of times a software that was developed once. Multipliers work both ways.

So, one day, the CEO has to face the facts. "We need a V2", he says. "We can invest, I have cash".

That's when the trouble begins.

Who Really Does The Maths?

Does the company have the right skills? Enough people? The right money?

As a consultant, I see that CEOs frequently don't run the maths, or maybe their calculator is bugged, or perhaps they are too optimistic.

New trendy technologies make them dream their product can be rewritten in 2 years maximum (it's always 2 years). For sure, they think current technology is much more productive that the old one they keep on using. And new developers can do so many pretty web stuff. The argument I hear also a lot is that the current software was refactored several times and so its global costs is much lower that the R&D team multiplied by the age of the software (generally around 10 to 20 years).

But, let's do the maths together: let's suppose the R&D development team was around 4 FTE during all those years. The gross product weight would be 60 men years (MY) for a team that worked 15 years on a product. Let's suppose 1/3 of the charge is not relevant and is corresponding to the refactorings. This leads to 40 useful MY.

Globally, a good developer in Europe costs around 100 k€ a year (with charges). That means that the software company should have approximately 4 M€ available to realize this project. This investment is generally quite risky.

40 MY of effort is possible in two years provided you have a team of 30 people (taking ramp-ups into account). For sure, the CEO of the company (or the CTO) is used to manage only 4 R&D people. So he rarely has the (project) management skills for such a big project. Let's suppose he reduces the team and targets 10 people for 4 years, he still has to train them for them to learn the business and to understand the existing product in order to be productive.

But where to find the right people? In IT service companies? This could be a bad move: IT service developer usually don't know how to produce good software; but they make it fast.

Most CEOs will choose one of those options:

- Be optimistic and start with his 4 developers using new technologies. Things will probably not progressed as expected.
- Outsource because a service company can convince the CEO that with 1M€, you can create 2M€ of software. Generally, the budget is consumed but the software is not produced (and not working).
- Believe software bandits or software magicians that will promise the CEO that with their code generation MDA approach, they can develop his V2 software in less than 2 years – provided he pays upfront.
- Develop a new product with a limited scope, for new customers, hoping to renew the old company success while fearing about the double team and the double recurring costs. But wouldn't it be a company startup within the company?
- Sell the company when it is still worth something and before it's too late.

If we do the maths, we can be worried.

Most of those CEOs will potentially try successively several options and fail. We can say they entered into the "V2 Vortex", a place where many software company go to die. Like the elephant cemetery, they cannot go back. Soon, they'll run out of cash and be obliged to find a more radical solution.

The V2 Vortex in Big Companies

In big companies, unfortunately, things are quite similar. Perhaps the application is much bigger (like a mainframe); perhaps it was not developed over 15 years but over 30; perhaps the development team is 100 people instead of 4; but the fact is here: they definitely are into the V2 Vortex.

Doing the maths is really frightening on their side. 30 years with 100 FTE leads to 3,000 MY of effort. Let's apply the 1/3 optimistic ratio: the application weights 2,000 MY... For sure, perhaps they don't have the cash problem. But they have tried to replace their systems 3 or 4 times, and all projects failed. Indeed, they never really understood why. This is too scary to look into it. They also tried several options and spent a lot of cash, and the amount spent is at the size of the ambition.

They, also, would like to get rid of the problem in 2 years, but that would mean 1,000 people in the project team... Nah, not possible.

IT service companies are not a big help with this kind of problem. Those applications are core business ones and they are connected to dozens of systems inside and outside their company. It is quite hard to even grasp the full picture.

They are in the V2 Vortex, and despite the fact that they're much bigger than the small software company we spoke about, the people in charge risk their job if they fail.

There is No Magic

I must say: there is no magic. I wish I could solve their problems with a lot of money but I cannot. However, there is a path - but they never like it.

"When you have applications with thousands of function points, you have to think about it as an asset", I say to them. Ok, it is old, they did not maintain it enough, it is bugged, they are locked in ancient design constraints and old technologies, yes, it is the same for everybody. But it is their asset.

Most of time, they can still sell it to customers (one day, they won't), or the application is still massively used (because it is core business). It still solves business issues. It still runs everyday for tons of people that can hate it while not being able to live without it.

So, as they would do with an old unmaintained asset that they have to keep, they have to *modernize* it. They cannot let it go. They have to do what they should have done, a long time ago. And it becomes urgent.

Rewrite by Parts and Refactor

New technologies brought new tools and new ways of managing legacy code. My advice is to upgrade to new tools, to new source management, to continuous build and integration, to pure local development workstations, to automated testing. The old code must absolutely be ported on recent compilers and OS.

Surprisingly, after all the previous failures, their development process is often the same old unproductive non agile process.

The hardest thing to do is to rewrite the software by parts, which implies code refactoring.

The final users of the applications will have to cope with a hybrid product for years, with some modernized parts and some old ones, the time for the team in place to modernize it completely.

Enterprise Architecture and Transition Architecture

Building The Functional Map, the Map to Other Systems and the Transformation Steps

Modernizing a complex software must be planned through the use of enterprise architecture techniques. As the system can be big in terms of functionality and/or can be connected to a lot of other systems, getting a complete map of the functionality and interfaces seems the best starting point. Archimate is the language for the task [2].

With Archimate, it is possible to represent the software progressive evolutions and the impacts on the user's processes. This is particularly important because some application modifications will have a serious impact on processes whereas others won't.

In terms of software, it is hard to migrate the core of an old application right away. Generally, the peripheric functions will be the easiest to replace. Basically, those functions often use data that, in the database, are not linked to any other data (leaves of the tree of tables).

By migrating function by function, and refactoring continuously the interfaces between the new part of the system and the old part of the system, the old part will become thinner while the new part will grow.

Transition Architecture

Technically, a transition architecture must be determined. The current one was supposed to be abandonned for a new one, but if we suppose the application will be composed by both technologies at the same time, it is crucial to have a generic way of moving functions from one environment to the other.

Identifying what process is creating the data and what processes are consuming the data are important knowledge to get in order to do a smooth migration. The migration should enable to transfer the control of data from the old system to the new one progressively.

Software is Made of People

The HR can be a problem. Let's face it: it is not so easy to convert people that were used for years to work in a certain paradigm in another. Some people will adapt and some won't. Moreover, developing in that phase of the life of the application requires skills that are generally not present in the original team.

Recruitment is generally getting on the critical path.

Because software is made of people. If people are closed, the software will be closed; if they are open, it will be open. If their people are messy, the software will be a mess; if they are structured, so will their software.

So, to create a great software, you need to hire great people. And, once you got them, you have to cherish them because bright people can be insecure. And you don't want to lose them.

This change is hard is many context:

- In small companies, it can bring a lot of issues due to feelings towards people (generally the original CTO is part of the founders and should be replaced);
- In big companies, the application manager, even if he took in charge for decades an application, must generally be changed in order for new motivated people to take over and do the work.

Tabula Rasa (Rewriting) Versus Transformation

Transformation Is Not Popular

In the software industry, we generally damage our software assets when maintaining them. Then comes a time when we must transform the big software.

Software transformation and refactoring is not very popular, for a lot of reasons:

- It requires a lot of work on the existing system, to really analyze what it does, how it was designed, and what are his good and bad points;
- It requires to enter into the fucntional details of the business, which many software engineers are reluctant to do, because they are more interested by technology than by functional aspects;
- It implies mastering two environments for a long period of time, and one of them may be very old and not very appealing on a résumé;
- It implies thinking at the software and IT architecture levels (3 and 4), even in a single application, which is not possible for many people [3].

For all those reasons, most people and managers prefer the "tabula rasa" solution. But, if we do the maths, we see that it is very often not an option.

The Rewriting Path

The rewriting path is not impossible, and it is sometimes the only option. It can be a success (even if many rewriting projects fail) but the conditions are numerous for it to succeed:

- The budget must be very big, and so must be the business case to get a ROI in a reasonable delay;
- The rewriting project must take into account the migration from the old system to the new one, the earlier possible in the project;
- The architecture of the V2 product should respect the semantics of the business, because may old systems do;
- The team must be great and work closely with the old system one;
- Evolutions should be limited in the old system;
- The architecture of the new system must enable parallel developments.

Conclusion

After decades of projects and auditing, most of the companies I saw facing the V2 vortex should have considered the transformation path rather than the rewriting path.

Notes

[1] - We will come back on technical debt in other parts of this book.

[2] - Archimate resources can be found here: http://www.opengroup.org/subjectareas/enterprise/archimate.

[3] - Refer to The Five Levels Of Conceptual Maturity for IT Teams.

(*June 2015, corrected November 2017*)