# Vision – Ontology - Action

## The model

Software development is a 3 dimensional operation, which are

- **V**ision: learning and designing the language and ideas of the customer;
- **O**ntology: an abstract, computer-ready form of our understanding;
- **A**ction: make it work – write code, configure components, get hardware, deploy.

They are really independent dimensions, and the direction of "growth" of a system goes "into" the space created by them: as we evolve, we should have a cleaner vision (domain understanding), better ontology (architecture) and better action (implementation). However, that is hard. Why?

Because we think of our software as a single, solid object in this space; and our implementation follows this understanding: we think of hardware requirements, platforms, toolkits, thousands or millions of source code lines, tons of man-years. The Artifact appears in this space, and as it grows, the harder it gets to change it.

Although, we should. A change on the concept level hits the model structure, that breaks current implementation, and when we have an implementation and show to the client, they will find new most important ideas that they forgot to mention in the first round, or misunderstandings that they told, but system creators did not really get. This constantly increases tensions in the system, until it can take no more and it starts to disintegrate.

## The playground

Taking this model, it is easy to draw a 3D picture from our IT environment. Close to and high on the V axis, we have domain expert systems like SAP or Word: they completely lock on a task. We don't think of them as something we want to integrate, about their internal model or implementation details. They "own" their field, encapsulate huge amount of knowledge and experience and work "as is". In new development, requirement management tools and tricks work on this dimension. On the O axis, we have modeling tools like UML editors or other DSL creators. They also live in their field, support modeling, but somehow alienated from the real life regardless of their followers who say they make everything much easier. At the A axis you find operating systems, platform dependent tools, etc. They "unveil" the capacity of an environment, you use them just as you like.

Each axis pair creates a plane for toolkits. On the V-O plane, you have domain modeling tools, frameworks that allow you to design a custom system with their concepts, like Salesforce. On the V-A plane, you have the high end domain toolkits, like image manipulation toolkits, GUI and platform frameworks like Jquery; that translate the features of a platform to a generic usage model. The O-A plane are the layer of "abstract programming": in Java, Spring or AspectJ, or servlet containers, SOAP implementations, and most of the popular acronyms. They make development itself relatively more efficient and robust by providing common solutions to generic problems.

This vision points out the obvious problems. Microsoft knows a lot about word processing, the guys at Oracle have millions of man-years in database development – but their knowledge is locked in their product. The same way, for new developments, we have requirements, but they are unclear, there is not force to modularize and reuse requirement segments. UML is here for decades, but it obviously failed to get to the same level in modeling as programming languages in action. The ontology management is not integral part of development, and therefore, it remained a promise, not a solution. The action plane is very hard: we have to choose deployment scenario (device only or server-client, fat or thin client), one or more platforms; our choice(s) drive the toolkit selection, and burn into the complete implementation. Any mistake at the very beginning can kill a project at the end - or what is even worse, create a zombie: a working-like money and effort sink.

If you try to avoid such mistakes in a development project, you tend move from the axis, don't do everything at home, but use toolkits from the planes. Unfortunately, that means separating from your own requirements and experience: you will force your client to accept what the toolkit can offer, distort your own understanding and bury all your findings into that or those toolkits. A quite high price in the long run, and just as risky decision as doing things in house – but at least you can replace your developers or outsource tasks to reduce the costs.

## What does Dust offer?

It breaks the fundamental, conceptual flaw: in Dust, an IT system is not a single Artifact. The requirements describe a segment in the VOA space, but it does not have to be completely filled by the volume of the system (codes, configurations, etc.), it can be almost empty and only behave like a solid object. When you develop a system in Dust, you actually:

- **Select the particles that you need.** A growing number of them will already be available in the Dust ecosystem; for the missing elements, you can easily create a completely hollow, "placeholder" or "mock" component, and clarify the actual services as you develop the system.
- **Place the particles to the required locations.** Create entities (user interface, database or email connector, workflow step, etc.), set up their attributes and connections. At this point, you can already see and demo the structure of your system, fully transparent to your clients because this networks "talks their language", where you see a component, they see a person, a role or an artifact in their domain. They can fix misunderstanding before you wrote a single line of code.
- **Make them work.** Of course, reused components bring their own behavior, so as you configured them, they will do their job. A GUI text box will display and change the attached data, the database connector will store and recall or search for your data, out of the box. You need to add your extra: the behavior of the elements in your clients' domain. Most of them, you can do by configuration, using the Process and Expression units of Dust, instead of actual, platform-dependent coding. Yes, that means you are compatible with all platforms that those units are implemented, and yes, they are part of Dust kernel, so...

The natural forces of system evolution are not broken by Dust – but they are cheated.

Your system will evolve: your clients will have new ideas after using the system, they will change the model, that will affect the materialization, and the changes will bring yet another new ideas. However, in Dust you never lose the transparency of your Vision, Ontology and Action. You have individual particles, or subnets of interconnected atoms to improve to follow the new needs; but the more critical a particle is (like a solid database, a GUI interconnection layer, a process or event management), the less likely that it is your job to fix it. Those parts are in the hands of the best experts on the planet, and it is very unlikely that they don't have a solid answer to any question you may have with their particles.

So, the tension caused by the changes do not crack the system, only hit exact particles and connections, and this can be fixed with transparent and manageable side effects. This lets your system grow and adapt organically to fit the external system it supports, for a very long time.