# Technical Debt, Data Engineering, and the OOD*

*one-off delusion

Dexter Taylor, Principal, Binary Machines LLC
dtaylor@binarymachines.io

There are two problems with "technical debt" as a term of art. The first is that it is a very often a euphemism for "poorly designed software", and as such it obscures that which is best made plain. The second is that it is only partly correct: it is not strictly a technical problem – about which, more later -- and it is only somewhat like a debt. Unlike ordinary debts, which can be tallied up by analyzing a ledger, there is no simple way of assessing technical debt. Quite often there is simply the growing awareness, at a particular firm, that a significant portion of the codebase does a poor job of controlling the costs and risks inherent in its own maintenance.

If we think of computer software as a system – an organized set of related components – then it should be evident that like any system, it has a given relationship with its operating environment. That environment is not limited to its host operating system or network; in the larger sense, computer software relates to a *usage* environment, a context in which humans (and other human-created systems) interact with the software and make demands on it.

To the extent that these interactions and demands vary over time, the software itself is subject to the impulse to change it. But change, in this context, always involves risk: the risk of introducing new errors, or degrading system performance, or both. And change always involves cost: the cost of assigning an engineer to do the work. In most firms, this is compounded by the opportunity cost of all the other work the assigned engineer is not doing.

So if we look at software as a system which experiences evolutionary pressure – the pressure to adapt to changing circumstances – then a necessary criterion for well-written software is that it minimizes the cost and the risk of adaptation over its service life. Deciding which adaptations are most likely to be required – forecasting the way in which the operating environment will change – and designing code with that forecast in mind without either **overengineering** or **optimizing too soon**, is fundamental to the art of software design.

Technical debt, as we have said, is somewhat like a debt because when engineers -- in the interest of saving time -- cut corners on their design and write brittle, unsightly code that is difficult to maintain, they are not *saving* time so much as they are *borrowing* it from the future.

Unless the software in question is discarded and/or rewritten after its first use, the time so borrowed will eventually be paid back, often at a crippling rate of interest.

Now software is almost never discarded after its first use; but business and engineering managers often deceive themselves that it will be. The mistaken belief that a problem will never arise again, or that a poorly-designed solution to it will not persist, is what we refer to as the **one-off delusion**.

It is worth noting that business managers lacking sufficient foresight (or insight) will often push even competent and experienced engineers to generate low-quality solutions very quickly. This is the sense in which, as we have said, technical debt is not entirely technical; it is in large part an outgrowth of a company's management culture. More and better technical talent can, over time, "pay off" technical debt -- but then companies overburdened by technical debt find it harder to attract and retain the very talent they need to pay it off. And a management structure in the grip of the one-off delusion will simply incur more technical debt over time.

To make matters worse, the one-off delusion is most seductive and reassuring in exactly the circumstances where it is most costly and damaging: that is, when the supposedly one-off problem is not customer-facing but internal in nature.

This is because (a) a successful firm is almost always outnumbered by its customers; (b) one's customers are not entirely predictable; and (c) one's potential customers are almost wholly *un*predictable. So it is easy to imagine that a novel problem or requirement coming from "outside the firewall" will appear again; perhaps even with enough regularity to create or destroy a competitive advantage (fortunes have been made and lost in just this way). Simply put, if one customer asks for something, others might as well. Good business managers know this instinctively, and that instinct works against the one-off delusion.

But data engineering, which is crucial in helping C-levels, managers, and analysts (not to mention data scientists) pilot the ship rationally, almost always involves deploying code which is *not* customer-facing. Very often the data engineer's only users are inside the company. This makes it easier for short-term thinking to take root and proliferate, and it is one reason why (for example) ETL stacks and data pipelines are pain points for companies attempting to correct systemic errors, streamline their analytics, or build a machine learning capability. These systems are often shot through with hacks, quick fixes, and workarounds which were created under the mistaken assumption that they would not last. The result is that tasks which should be easy become quite difficult, and difficult tasks nearly impossible. Routine changes take longer than they should to implement and are more likely to introduce errors. Morale suffers, undermined by the perception of waste and futility. Programmers who are content to grind out mediocre code at high speed may endure this quite well, but your most talented and imaginative engineers will drift – sometimes toward the exits.

When customer-facing software violates SLA or becomes overly costly to maintain; when a company finds its trouble tickets piling up or new features backlogged -- when these things happen, warning lights go on all over the organization, which is oriented (as it should be) toward acquiring and retaining customers and keeping revenue flowing. In this context, a negative trend, like the prospect of hanging, concentrates the mind.

But the costs imposed by poor back-office engineering practices, by contrast, are much easier to ignore, especially when resources are constrained and deadlines are tight. In these straits it is tempting to look at an unusual analytics need, or business rule, or accounting formula, or data format, and see it as a one-off. Sometimes, rarely, it is. Much more often, it is a warning.

It is deep inside the firewall where you can more readily convince yourself that there are no threats. Everyone has seen the fanciful ancient maps of the world which bear, at the edges, the inscription “Here be dragons”. The implication is that there are no dragons at the center. Those of us in IT, and especially in data engineering, know better than that.

Dexter Taylor
Principal, Binary Machines