Regress or Progress?

Seeing Good Software Engineering Ideas Through

Hakan Erdogmus

In the last edition of From the Editor (“Déjà Vu: The Life of Software Engineering Ideas,” January/February 2010), I wrote about how modern software engineering ideas evolve. I represented an idea’s maturation life cycle from conception to streaming in terms of nine states, interactions with the life cycle of other related ideas, and regressive loops that preempts an idea’s normal progression and revert it to a previous state. In this edition, I discuss the levers that help prevent reversion, or at least premature reversion, and push a worthwhile idea forward toward the streamed state. These levers work only for good ideas, or those that have genuine merit, and before the idea reaches the streamed state. Once an idea reaches that point, we don’t have much control; short attention spans are inevitably diverted elsewhere.

Idea, Bundle, and Context

In “Déjà Vu,” I defined the concepts of bundle and context. Reviewing the differences among “idea,” “bundle,” and “context” will set the stage for discussing the levers that underlie my model. The context is “the general environment in which an idea is positioned.” It includes “problems being addressed, counterideas, accelerators, decelerators, target audience, surrounding technologies,” and “relevant technical, social, and economic conditions.” A bundle is “the immediate context that an idea’s advocates deem necessary to make the idea operationally viable and allow it to be applied in practice.” It’s the “collection of vital complementary and synergistic” elements that “the proponents think will make the idea work.”

The distinction between an idea and a bundle is sometimes a matter of granularity and abstraction level. If the idea is the central phenomenon of interest, then the bundle is its larger packaging. When we decompose an idea further, it becomes the bundle for its constituents. However, a mere change in perspective doesn’t separate the context from the bundle. An idea’s proponents can’t exert any control over the context, but they determine the bundle. Contexts naturally tend to be messy because the world surrounding us is complex. Table 1 summarizes the differences between a context and a bundle.

Five Levers

Figure 1 lists the five levers that help advance worthwhile ideas, along with the levers’ corresponding life-cycle states.

Lever 1: Unbiased Reflection

An objective evaluation of an idea’s previous incarnations relative to the current context fosters learning that helps avoid recurring, unproductive traps. Two unproductive traps are reversible mellowing...
and pendulum swings. Reversive mellowing happens when an oversold idea creates a contrarian reaction during the mellowing state, the result of which is a competing idea or an opposite school of thought taking over. If the overtaking idea suffers the same fate, a pendulum swing occurs, with opposing approaches replacing each other only to suffer reversion themselves.

Unbiased reflection entails evaluating an idea’s pedigree, or its lineage involving previous incarnations and conceptual predecessors, and its antipedigree, or the lineage of competing approaches, to understand why an idea is worthwhile now and how the current context differs from former contexts. Awareness and articulation of these differences not only prevent repeating previous mistakes but also help position the idea uniquely through effective branding (which I’ll discuss later).

What has changed in the environment? How is the idea’s current bundling differentiated from those of the pedigree? In what ways is the current bundling advantageous? In what ways is it disadvantageous? How successful were the previous incarnations? How successful were any of the counter-ideas under specific circumstances? Why haven’t the previous incarnations taken off? These questions lie at the heart of unbiased reflection.

More often, an idea’s proponents succumb to biased reflection in the form of either false differentiation or false association. In fact, both tactics work, but the effects are only temporary. False differentiation is seeing and selling in a new light an old idea that is revisited in essentially the same context as its previous incarnations. The proponents try hard to dress the idea up in new clothes, focus on minor technicalities to differentiate it from the pedigree, or just resort to using a catchier terminology. False association is a case of 20/20 hindsight, the kind I addressed in “Novelty in Sameness” (IEEE Software, May/June 2007) that involves reinterpreting old propositions outside their original contexts to push a related but more modern idea. False association occasionally manifests itself as a claim along the lines “when pioneer Joe advocated X some 30 years ago,

![Figure 1. The life cycle and underlying adoptability levers of modern software engineering ideas.](image-url)

Table 1

<table>
<thead>
<tr>
<th>Context</th>
<th>Bundle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large</td>
<td>Small</td>
</tr>
<tr>
<td>Elements may have many degrees of separation</td>
<td>Elements have a few degrees (single degree?) of separation</td>
</tr>
<tr>
<td>Not necessarily cohesive</td>
<td>Cohesive</td>
</tr>
<tr>
<td>Messy</td>
<td>Neat</td>
</tr>
</tbody>
</table>

Table 1: Context vs. bundle

From the Editor
he actually meant Y, and you’ll see the evidence if you read between the lines in such and such paper.” Curiously, sometimes X is the opposite of Y.

**Lever 2: Neat Bundles**

The overarching quality of a bundle is neatness. A neat bundle is small (it contains a few elements), cohesive (its elements are closely related, highly synergistic, and complementary), essential (it has low redundancy), and loosely coupled, both internally (its elements can be understood and applied independently, with minimal reliance on each other) and externally (it has few dependencies on the context). While cohesion captures relatedness among elements, coupling captures their reliance, or dependence, on each other. To avoid the intrinsic tension between cohesion and coupling, I’m assuming that being codependent is distinct from being synergistic or complementary.

But why is neat bundling important? The question leads to what I term the adoptability barrier conjecture. An original means for solving a problem should be more cost effective than the standard way of solving that problem. More specifically, the total effort required to understand, learn, and apply the new means of solving the problem’s expected future instances must be lower than the total effort required to solve those instances by the alternative or de facto means. Hence, if \( N \) is the expected number of instances of the problem to be solved and the benefits of solving the problem by whatever means remains constant, we can express the adoptability barrier conjecture as

\[
\text{LearningCost}(\text{NewMeans}) + N \times \text{ApplicationCost}(\text{NewMeans}) < N \times \text{ApplicationCost}(\text{Old Means}).
\]

The right-hand side of the inequality, \( N \) times the average application cost of solving the problem by the old means, is the adoptability barrier. As \( N \) gets larger and \( \text{ApplicationCost}(\text{NewMeans}) \) drops, \( \text{LearningCost}(\text{NewMeans}) \) ; the usually one-time learning cost of the new means) can be more easily amortized over time to make the new solution cost effective relative to the status quo.

Some context-related forces that are outside our direct control, such as changing technology and turnover, may prevent amortization through recurring learning costs. And uncertainty in the problem’s expected number of instances may void any amortization guarantee we might have vis-à-vis the new means. Fortunately, the bundle can be controlled.

Adoptability favors neat bundles because neat bundles are likely to incur lower learning and application costs. As bundles get bigger, they also get messier, more complex, and difficult to learn and apply. The messiest bundles contain everything but the kitchen sink.

**Lever 3: Effective Branding**

Effective branding involves both choosing a representative vocabulary that is aligned with the context and rational positioning of the idea based on unbiased reflection. However, it’s not about dressing up an idea with excessive jargon and grandiose claims to make the problem seem more complex than it is or to give the solution the illusion of originality. Although such strategies might temporarily inflate an idea’s popularity, as in biased reflection, their effects aren’t long-lasting enough to allow the idea to be streamed before reversing forces catch up with it.

While adoptability favors neat bundles, neat bundles can be perceived as simple-minded and narrowly scoped, and consequently harder to sell to a broad audience. Such perceptions create a tension between effective branding and neat bundling. But not all of this tension is given rise by mere perception: the reality of dwindling funding and the tendency of funding programs to emphasize large, consolidated projects also come into play. The need to be given serious consideration by prospective funders drives good ideas to be wrapped in messier and messier bundles in an inevitable effort to cater to multiple interests and combine otherwise thinly spread resources. In such situations, marketability trumps neatness and effective branding takes the shortest path to much needed resources.

**Lever 4: Easy Sandboxing**

Sandboxing is gauging an idea’s feasibility in a safe and cheap nonproduction environment. It’s an important learning strategy widely used in the testing state of an idea’s life cycle.

Messy bundles tend to feature overlapping concepts with fuzzy boundaries and multiple, complex, interdependent technology stacks. The multiplicity, complexity, and codependence of the prerequisites make sandboxing the underlying idea difficult. The outcome is a much steeperened learning curve and, ultimately, strained or preempted adoption. Therefore, easy sandboxing is often a critical lever for adoptability.

**Lever 5: Optimal Extraction**

The last lever concerns the way in which an idea is spun off, or extracted, from its bundle toward the end of its growth-and-decline phase to continue its journey toward a stable existence. The extraction’s form, size, and timing affect the idea’s remaining life. Many software engineering ideas are wrapped up in larger bundles only to be unwrapped later to relive a life of their own. An obvious example is the extraction of test-driven development from its famously controversial former bundle, Extreme Programming.

We can extract an idea from its former bundle in different ways:

- as is, nearly as originally intended before it was integrated into the bundle;
- in a distilled form, freed from any appendages that have failed to work in the
field or that used to bridge the idea to other elements in the former bundle;

- in a modified form, with tweaks that have been demonstrated to make it more applicable;
- in a more concrete form, for example, as an implementation, possibly embedded in a tool or in a related technology stack; or
- in a more generalized form, abstracting it away from any details that unnecessarily limit application.

The best form depends on the cumulative feedback received during the testing state of the idea.

The timing and size of the extraction are other factors. The extraction is rightsized if the idea isn’t incidentally oversimplified, stripped of some of its essential elements, overgeneralized, overextended, or diluted to the point where it blends into the background context before getting a chance to stick with a distinct identity. The extraction is well timed if the idea has been tested long enough within its former bundle to receive meaningful feedback, but before being flattened for having lingered too long in the testing state.

**Are the Five Levers All that Matter?**

No. As in pretty much anything that governs our industry, numerous other contextual factors play a role in the maturation, acceptance, and adoption of good software engineering ideas. These factors notably include the availability of evidence, but the value of evidence in itself depends on a variety of other underlying factors: the purpose of the application, the idea’s virulence, the type of evidence, and the risk-taking characteristics of individual receptors, to name the most central ones. Because I can’t do justice to the role of evidence in the space allotted to me, I must stop here.

I have built on the proposition that modern software engineering ideas follow an incremental, reversible, and threaded maturation process before they enjoy a stable, respectable existence. Although the five levers I’ve presented carry the caveat of being neither sufficient nor necessary for acceptance and adoption, I believe they’re generally applicable. Barring a safe exception, I deliberately avoided giving specific examples, thus conveniently leaving that task to the reader’s imagination.

So, how do you position your favorite trend with respect to the five levers? Is its bundle neat or messy? To which levers is the trend particularly sensitive? Which ones have significantly been affecting its adoption? And finally, is there anything to be done about the remaining hurdles? Send your comments to me at hakan.erdogmus@computer.org.