Is Detailed Design Anti-Agile?

Description

In the Beginning . . .

In days of yore, systems development projects were front ended with laborious requirements engineering and design tasks. This made sense then because development was labor-intensive, time-consuming and expensive. Changes to the scope or design of a solution mid-development increased the likelihood of errors and incremental time and expense. In recognition of this, traditional Waterfall project management was applied, which created impediments to modifying the product definition once its development had begun. Changes were strenuously resisted.

A couple of factors conspire to undermine the Waterfall approach. First, it depends on users conceptualizing their needs in detail, ex ante, and getting them right. That explains how requirements engineering got its nameâ€”IT experts were expected to lead users (Product Owners and Managers in todayâ€™s lingo) through the process of ideating and defining what their solution should be before any of it is designed or built. Ideally, the engineers and architects can steer users away from the frivolous and toward the feasible; however, to the degree that neither side fully understands the other, mistakes and omissions are common. Secondly, as projects took so long to complete then, the world changed, and aspects of the product as originally designed were obviated or invalidated.

Enter Agile . . .

Experience has revealed some rather large holes in the assumptions underlying the traditional waterfall approach. The need to be able to change course while a product or solution is in development necessitated some new thinking: ***enter Agile***. What supposedly made Agile work was iterative development, in which elements of a solution are developed in short sprints and integrated to allow users to see and touch them at frequent intervals. If seeing something tangible sparks different thinking about the solution, Agile provides an opportunity to change course before too much time and effort have been spent. This has had some ameliorative impact. Solutions seem to be trending toward meeting more of the actual requirements, including those that surface or morph between their initial specification and their implementation.

Two areas of emphasis in Waterfall project management are predictability and risk management. Project execution success is measured against the triple constraint. The team is expected to deliver the **defined scope** within the **time** and **budget** allotted for it. This is, in general, quite a reasonable goal. No one wants to sign up for an unscripted thrill ride resulting from unmitigated execution risks. One of most pernicious of these risks, it should be noted, is evolving scope, resulting from incomplete requirements analysis or a rapidly evolving environment. Missing any of the three parameters of the triple constraint is considered bad form and has been thought to indicate something about the flawed character of project managers on whose watch it occurs.

Unfortunately, itâ€™s all anti-agile. If you attempt a Scrum implementation within the envelope of a project managed using a Waterfall approach, you get Water-Scrum-Fall, which I have written about in previous articles. Given how antithetical Agile and traditional Waterfall PM are to each other, some rethinking is in order.

Against this backdrop, letâ€™s consider what an ideal organizational stance relative to agile digital product development should look like. It will have:

- An enlightened agile company that treats its products as elements of a portfolio, diversified along a number of dimensions, and which allows business units to allocate development funds at their discretion. Experimentation for both product development and product evolution is encouraged and there are no reprisals for well-conceived experiments that do not result in marketable products with high returns on investment.
- Involved Product Owners and Managers that are tightly integrated with the development and delivery teams.
- Highly knowledgeable development teams with expert Agile development and DevOps capabilities.
- Constructive culture and supportive leadership norms.

But, Agile isnâ€™t a Panacea . . .

You would expect that this would position you to achieve optimal results but there are some things that traditional Agile never provided for and which merit more attention than they often getâ€”technical debt (of various types) and data architecture. Why should this be? The original Agile Manifesto was written when monolithic applications were common and when a single team, albeit a large one, was building elements of an entire solution concurrently. As technology evolved toward solutions constructed as integrated constructs comprised of many parts, breaking the team up to have subsets focus on highly cohesive, loosely coupled components is becoming increasingly common. This has advantages and disadvantages and not accounting for them can have severe consequences for your work products.

A constant concern in systems development is managing trade-offs. **Optimizing for rapid delivery** in earlier times meant intensive design effort, fixed scope and strict restrictions on change. On the other hand, **accommodating evolving requirements** generally meant work slowing or stopping while the solution design was revised, and the remaining development replanned. It also meant that a projectâ€™s cost and schedule estimates were likely to be transcended. **Providing for flexibility to address future requirements** meant spending more time on upfront design so that the most likely foreseeable new ones could be accommodated as minor changes rather than major revisions.

Current technology, such as low-code/no-code development tools, open-source software, cloud infrastructures, SaaS services, containerized software, serverless back-end functions and managed database services resets the optimal balance between designing and doing in todayâ€™s development paradigms. Trade-offs must still be considered and addressed, however. Consider a path a new requirement might take between ideation and realization:

- **Ideation**: An initial definition of a product, a feature or a function is fleshed out to the point that it can be turned over to the architects.
- **Design**: The developers create a preliminary design for a component or components to enable the product or feature.
- **Estimation**: Developers produce a preliminary estimate of what they believe it will take them to build it based on the rough design.
- **Qualification**: The design and estimate are reviewed to identify whether the design adheres to architectural standards, or it is redundant with existing components that could be reused. Interdependencies and interconnections with existing or planned components are identified and analyzed to ensure that they are well-understood.
- **Prioritization**: The task(s) of implementing the component(s) are placed into the backlog of waiting work and slotted into a queue. They are prioritized based on any of a number of considerations, such as their being precursors to other components that are dependent on them or their being elements of high-value, high-visibility product features.
- **Development**:The components are developed and **unit-tested**. When they pass all their unit tests, they are checked into the productâ€™s repository.
- **Initial Deployment**: The components, and any others on which they may be dependent, are built and integrated into an executable package and deployed to a test environment where they can undergo **integration testing**.
- **Internal Review and Evaluation**: Components that pass integration tests are made available to POs, PMs, user representatives and members of the development team to review and evaluate.
- **Disposition**: Based on feedback from the evaluation, one of four things can happen:
    - **Revision**: The team can return it to the design phase, refine the componentâ€™s definition and modify the software to meet the revised definition. If the interim product fails to perform properly but does not appear to have behavioral deficits, it might just be sent back to development to fix programming errors.
    - **Limited Release:** The product can be released as a Beta or MVP version and made available to a limited set of external users.
    - **Production Release:** The product can be deemed as ready to release to the market.
    - **Termination:** If the team determines that the product or feature offers no feasible path to provide sustained value, then it may be discontinued, and the resources dedicated to it reassigned.
- **Product Monitoring:** The commercialized productâ€™s reception and market performance are monitored to inform the ongoing evolution process. At any time that the changes to the product are deemed to be necessary or desirable, it is returned to the relevant point in the development cycle.

Looks rather waterfallish, doesnâ€™t it? In fact, itâ€™s nearly impossible *not* to employ a fairly linear path when building individual software components. You have to design them before you build them, you have to test them before you release them and so on. The major difference between Agile and traditional Waterfall is how it supports iterative refinement of the product as an *expected characteristic*

of the process, rather than a *result of a mistake or omission in requirements engineering*. Another thing that you may notice is that architecture in the process, above, doesnâ€™t get addressed until qualification. This is a significant omission in Agile as it was initially described and is frequently practiced.

Balance in All Things . . .

Letâ€™s look a little harder at Agile and its place in the enterprise ecosystem. **The true goal of agile development is to enable you to react as quickly as possible to new opportunities and threats *without creating unnecessary technical debt*.** Accumulated debt impedes your ability to react and, as you amass more and more of it, it can slow you to a crawl. Conversely, identifying and reusing existing components, accelerates your response at the same time that it lowers your costs and reduces your risks. After all, *not* building something that you donâ€™t have to saves time and money and eliminates the potential that it will take longer to build than you thought or that you will break something that youâ€™re then going to have fix.

Designing for reusability costs time and money but it must be viewed as an investment. Usually, it is more expensive and time-consuming than designing and building just to meet an immediate need. The payoff for this usually occurs in the future, when you can reuse something you have instead of building something new or when you can minimize the customization requirements of the next iteration of what youâ€™ve built. Therefore, you have trade-offs to consider and a major determinant that should inform your choices is *architecture.* Your architecture group(s) should play a significant role in your Agile development for just this reason.

Architecture vs. Design

There has been a lot of discussion for quite a while about whether these two things are the same, or not. Design is an element of architecture, but they are intended for two different purposes. Architecture defines the structure of what is to be built while design provides a detailed definition for exactly how it will be built. Architecture is WHAT; design is HOW, and I have observed before that when HOW precedes WHAT, trouble usually follows.

The difference between architecture and design, relative to the process of building a solution, is significant. Hereâ€™s why:

- Design depends on architecture. Some developers leap from requirements to design without explicitly defining and validating the solution architecture. There is an architecture here, but it is implicit and may or may not conform to enterprise standards, exploit reusable artifacts or integrate well with other elements of the enterpriseâ€™s overall architecture.
- Design, being more granular and detailed, takes more time and effort than solution architecture does.
- Given the effort that goes into them, developers often fall in love with their solution designs, which creates inertia that impedes their willingness to revise and refactor them as development progresses and more information is gathered. However, going down a wrong path and reversing course wastes time and slows delivery.

This is the nexus of a problem: **if your implicit architecture is wrong, the chances are good that your design is flawed and if you are attached to it, you will resist changing it**. You begin the process of building a new solution with the least information you will have over the course of the implementation, yet you tie yourself to decisions you may have made at that time. This is anti-agile.

What do you need, then? You need just in time design! You should start each development initiative by attaining a clear understanding of the business case for building a solution and the Objectives and Key Results (OKRs) which will be used to assess the success of the work. You should also obtain a clear understanding of the assumptions underlying the requirements and OKRs and the hypotheses that the POs and PMs are looking to confirm or refute in the course of implementation.

Given this, the PMs and solution designers should work with the architects to define the architectural patterns and elements that will be employed in the solution and how these elements will be implemented. Some of them will be existing artifacts that can be reused, some may be purchased, some may be custom built and so on. What is crucially important is that how the solution will integrate with the enterpriseâ€™s architecture will be understood and accommodated in the design when it is defined. Then, you should identify which elements of each of the major architectural components of the solution are required early in the development process to allow you to validate your assumptions or confirm your hypotheses. These elements should be prioritized in your implementation plan.

The agility in Agile comes from it being a low-impact way to enable and accommodate evolving product or solution definitions as solutions are being built and information about them is accumulating. It implicitly accounts for and accommodates the incomplete state of knowledge that exists at the outset of an implementation effort. Overdesigning at that point undermines this entirely. It wastes time and effort up front to perform counterproductive analysis that ultimately impedes evolution.

It is well understood that the earlier in development an error or omission occurs and the later it is found, the more it will cost to fix. This was true in the early days of automation, and it is still true today. Defining user stories and use cases in painstaking detail at the outset of an implementation project in order to plan it has the potential to waste time and set the initiative on a course from which it could have to retreat and retrench. My advice is to establish the solution architecture first and perform detail design work when you get to the point of implementing major elements. This will allow you to exploit the most current available information, which will include at that point everything you will have learned from implementing the other components that already exist.

Now, thatâ€™s agile!

Date Created
2022/05/05
Author
howardmwiener